# Linux 的 5 种主要 I/O 模式

## 同步阻塞i/o(BIO)

应用进程向内核发起 I/O 请求，发起调用的线程一直等待内核返回结果。一次完整的 I/O 请求称为BIO（Blocking IO，阻塞 I/O），所以 BIO 在实现异步操作时，只能使用多线程模型，一个请求对应一个线程。但是，线程的资源是有限且宝贵的，创建过多的线程会增加线程切换的开销。

## 同步非阻塞i/o(NIO)

应用进程向内核发起 I/O 请求后不再会同步等待结果，而是会立即返回，通过轮询的方式获取请求结果。NIO 相比 BIO 虽然大幅提升了性能，但是轮询过程中大量的系统调用导致上下文切换开销很大。所以，单独使用非阻塞 I/O 时效率并不高，而且随着并发量的提升，非阻塞 I/O 会存在严重的性能浪费。

## I/O多路复用

多路复用实现了一个线程处理多个 I/O 句柄的操作。多路指的是多个数据通道，复用指的是使用一个或多个固定线程来处理每一个 Socket。select、poll、epoll 都是 I/O 多路复用的具体实现，线程一次 select 调用可以获取内核态中多个数据通道的数据状态。多路复用解决了同步阻塞 I/O 和同步非阻塞 I/O 的问题，是一种非常高效的 I/O 模型。

## 信号驱动I/O

信号驱动 I/O 并不常用，它是一种半异步的 I/O 模型。在使用信号驱动 I/O 时，当数据准备就绪后，内核通过发送一个 SIGIO 信号通知应用进程，应用进程就可以开始读取数据了。

## 异步I/O

异步 I/O 最重要的一点是从内核缓冲区拷贝数据到用户态缓冲区的过程也是由系统异步完成，应用进程只需要在指定的数组中引用数据即可。异步 I/O 与信号驱动 I/O 这种半异步模式的主要区别：信号驱动 I/O 由内核通知何时可以开始一个 I/O 操作，而异步 I/O 由内核通知 I/O 操作何时已经完成。

# Netty的io模型

Netty 的 I/O 模型是基于非阻塞 I/O 实现的，底层依赖的是 JDK NIO 框架的多路复用器 Selector。一个多路复用器 Selector 可以同时轮询多个 Channel，采用 epoll 模式后，只需要一个线程负责 Selector 的轮询，就可以接入成千上万的客户端。

在 I/O 多路复用的场景下，当有数据处于就绪状态后，需要一个事件分发器（Event Dispather），它负责将读写事件分发给对应的读写事件处理器（Event Handler）。事件分发器有两种设计模式：Reactor 和 Proactor，Reactor 采用同步 I/O， Proactor 采用异步 I/O。

# Netty整体架构

Netty的结构一共分为三块

## core核心层

它提供了底层网络通信的通用抽象和实现，包括可扩展的事件模型、通用的通信 API、支持零拷贝的 ByteBuf 等。

## protocol support 协议支持层

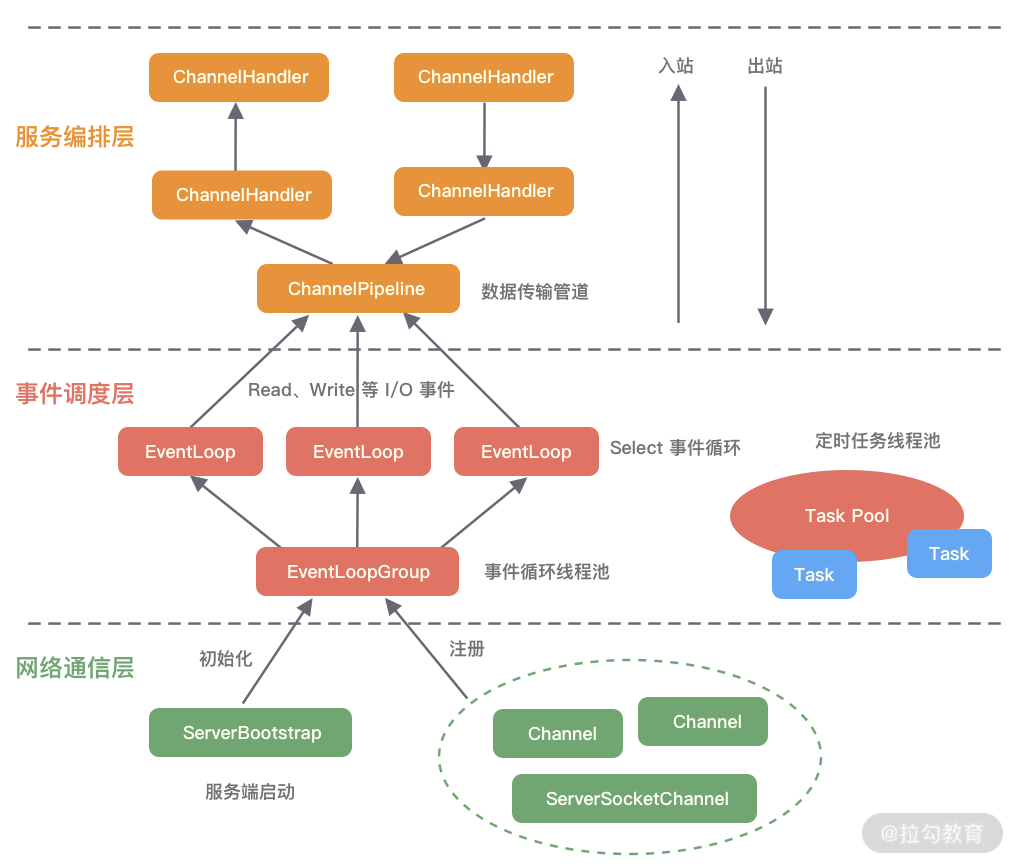
协议支持层基本上**覆盖了主流协议的编解码实现**，如 HTTP、SSL、Protobuf、压缩、大文件传输、WebSocket、文本、二进制等主流协议，此外 Netty 还支持自定义应用层协议。Netty 丰富的协议支持降低了用户的开发成本，基于 Netty 我们可以快速开发 HTTP、WebSocket 等服务。

## transport server 传送服务层

传输服务层提供了网络传输能力的定义和实现方法。它支持 Socket、HTTP 隧道、虚拟机管道等传输方式。Netty 对 TCP、UDP 等数据传输做了抽象和封装，用户可以更聚焦在业务逻辑实现上，而不必关系底层数据传输的细节。

# Netty的逻辑架构

Netty 的逻辑处理架构为典型网络分层架构设计，共分为网络通信层、事件调度层、服务编排层，每一层各司其职。



## 网络通信层

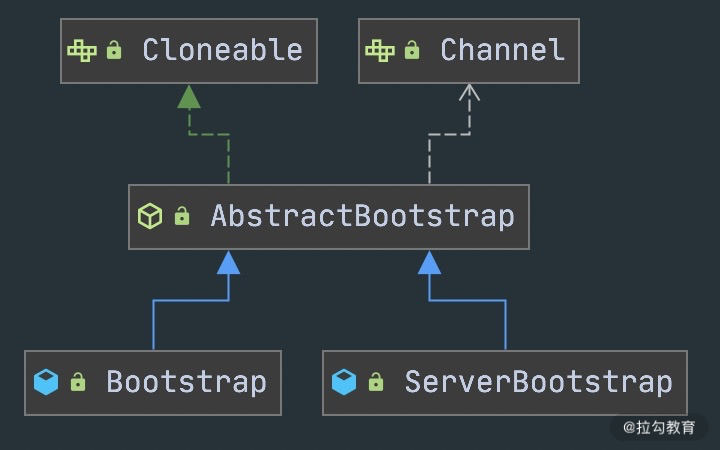
网络通信层的职责是执行网络 I/O 的操作。它支持多种网络协议和 I/O 模型的连接操作。当网络数据读取到内核缓冲区后，会触发各种网络事件，这些网络事件会分发给事件调度层进行处理。

网络通信层的核心组件包含BootStrap、ServerBootStrap、Channel三个组件。

### Bootstrap&ServerBootStrap

Bootstrap 是“引导”的意思，它主要负责整个 Netty 程序的启动、初始化、服务器连接等过程，它相当于一条主线，串联了 Netty 的其他核心组件。

Netty 中的引导器共分为两种类型：一个为用于客户端引导的 Bootstrap，另一个为用于服务端引导的 ServerBootStrap，它们都继承自抽象类 AbstractBootstrap。



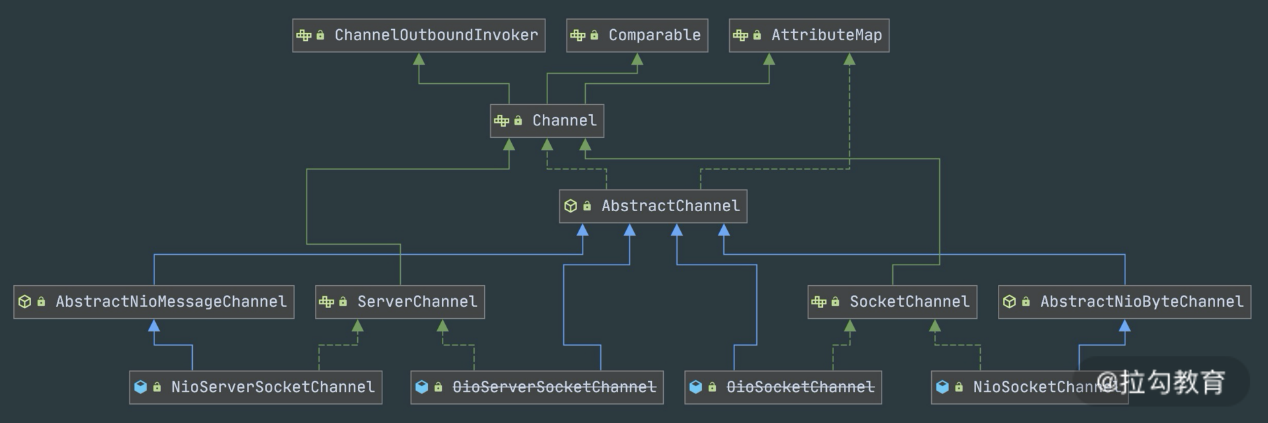
Bootstrap 和 ServerBootStrap 十分相似，两者非常重要的区别在于 Bootstrap 可用于连接远端服务器，只绑定一个 EventLoopGroup。而 ServerBootStrap 则用于服务端启动绑定本地端口，会绑定两个 EventLoopGroup，这两个 EventLoopGroup 通常称为 Boss 和 Worker

每个服务器中都会有一个 Boss，也会有一群做事情的 Worker。Boss 会不停地接收新的连接，然后将连接分配给一个个 Worker 处理连接。

### Channel

Channel 的字面意思是“通道”，它是网络通信的载体。Channel提供了基本的 API 用于网络 I/O 操作，如 register、bind、connect、read、write、flush 等。Netty 自己实现的 Channel 是以 JDK NIO Channel 为基础的，相比较于 JDK NIO，Netty 的 Channel 提供了更高层次的抽象，同时屏蔽了底层 Socket 的复杂性，赋予了 Channel 更加强大的功能，你在使用 Netty 时基本不需要再与 Java Socket 类直接打交道。

下图是 Channel 家族的图谱。AbstractChannel 是整个家族的基类，派生出 AbstractNioChannel、AbstractOioChannel、AbstractEpollChannel 等子类，每一种都代表了不同的 I/O 模型和协议类型。常用的 Channel 实现类有：



NioServerSocketChannel 异步 TCP 服务端。

NioSocketChannel 异步 TCP 客户端。

OioServerSocketChannel 同步 TCP 服务端。

OioSocketChannel 同步 TCP 客户端。

NioDatagramChannel 异步 UDP 连接。

OioDatagramChannel 同步 UDP 连接。

当然 Channel 会有多种状态，如连接建立、连接注册、数据读写、连接销毁等。随着状态的变化，Channel 处于不同的生命周期，每一种状态都会绑定相应的事件回调，下面的表格我列举了 Channel 最常见的状态所对应的事件回调。

事件 说明

channelRegistered Channel 创建后被注册到 EventLoop 上

channelUnregistered Channel 创建后未注册或者从 EventLoop 取消注册

channelActive Channel 处于就绪状态，可以被读写

channelInactive Channel 处于非就绪状态

channelRead Channel 可以从远端读取到数据

channelReadComplete Channel 读取数据完成

有关网络通信层我就先介绍到这里，简单地总结一下。**BootStrap 和 ServerBootStrap 分别负责客户端和服务端的启动，它们是非常强大的辅助工具类；Channel 是网络通信的载体，提供了与底层 Socket 交互的能力**。那么 Channel 生命周期内的事件都是如何被处理的呢？那就是 Netty 事件调度层的工作职责了。

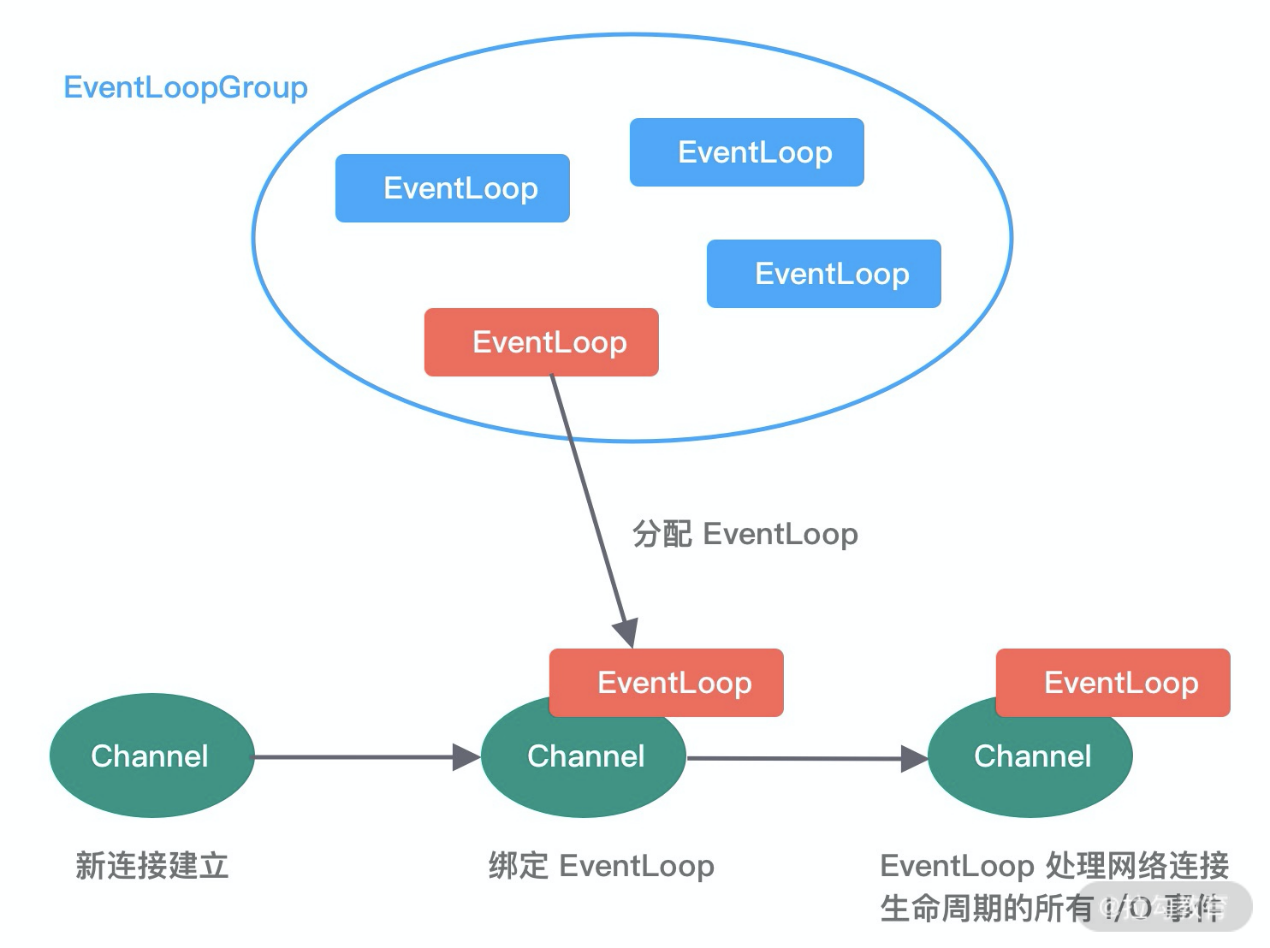
## 事件调度层

事件调度层的职责是通过 Reactor 线程模型对各类事件进行聚合处理，通过 Selector 主循环线程集成多种事件（ I/O 事件、信号事件、定时事件等），实际的业务处理逻辑是交由服务编排层中相关的 Handler 完成。

事件调度层的核心组件包括 EventLoopGroup、EventLoop。

EventLoopGroup & EventLoop

EventLoopGroup 本质是一个线程池，主要负责接收 I/O 请求，并分配线程执行处理请求。

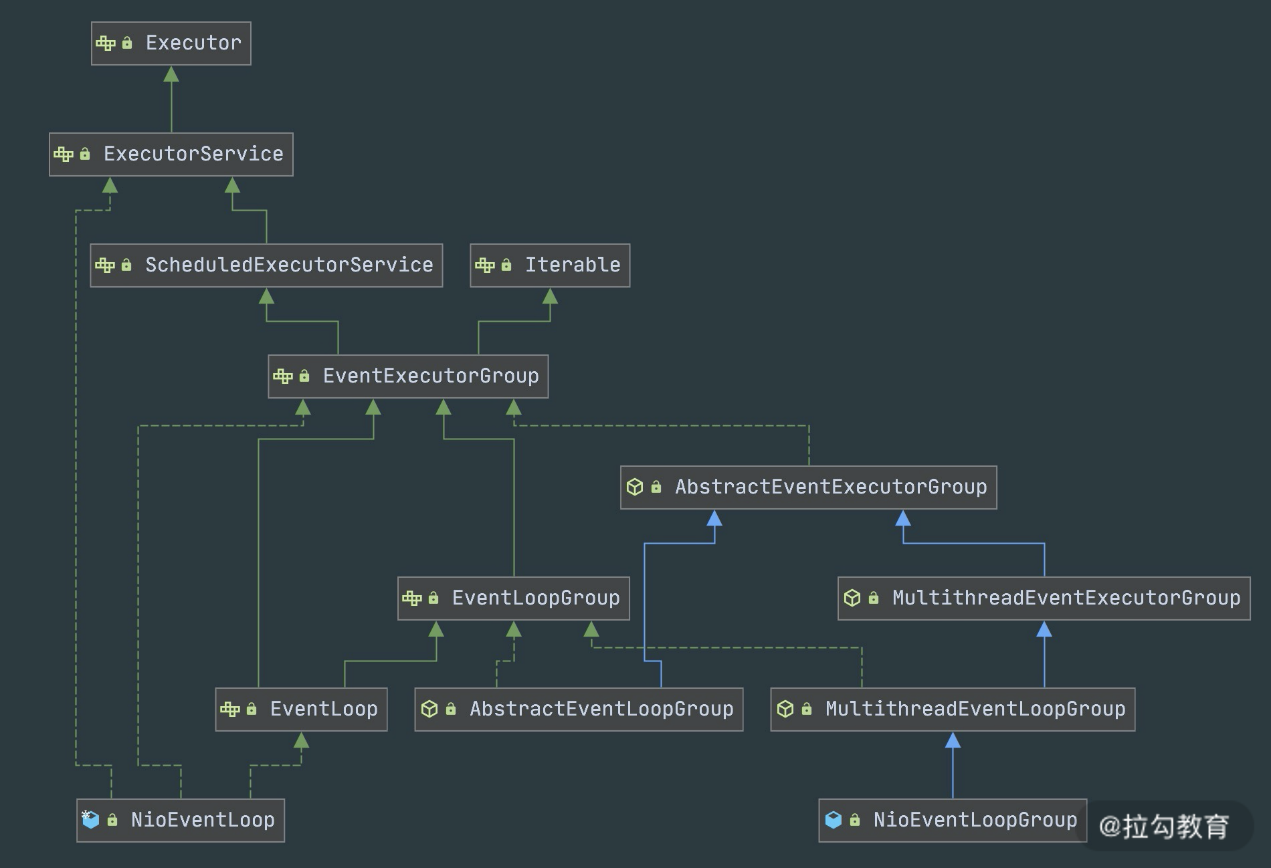


EventLoopGroup、EventLoop、Channel 的几点关系。

一个 EventLoopGroup 往往包含一个或者多个 EventLoop。EventLoop 用于处理 Channel 生命周期内的所有 I/O 事件，如 accept、connect、read、write 等 I/O 事件。

EventLoop 同一时间会与一个线程绑定，每个 EventLoop 负责处理多个 Channel。

每新建一个 Channel，EventLoopGroup 会选择一个 EventLoop 与其绑定。该 Channel 在生命周期内都可以对 EventLoop 进行多次绑定和解绑。



EventLoopGroup 的实现类是 NioEventLoopGroup，NioEventLoopGroup 也是 Netty 中最被推荐使用的线程模型。NioEventLoopGroup 继承于 MultithreadEventLoopGroup，是基于 NIO 模型开发的，可以把 NioEventLoopGroup 理解为一个线程池，每个线程负责处理多个 Channel，而同一个 Channel 只会对应一个线程。

**EventLoopGroup 是 Netty Reactor 线程模型的具体实现方式**，Netty 通过创建不同的 EventLoopGroup 参数配置，就可以支持 Reactor 的三种线程模型：

单线程模型：EventLoopGroup 只包含一个 EventLoop，Boss 和 Worker 使用同一个EventLoopGroup；

多线程模型：EventLoopGroup 包含多个 EventLoop，Boss 和 Worker 使用同一个EventLoopGroup；

主从多线程模型：EventLoopGroup 包含多个 EventLoop，Boss 是主 Reactor，Worker 是从 Reactor，它们分别使用不同的 EventLoopGroup，主 Reactor 负责新的网络连接 Channel 创建，然后把 Channel 注册到从 Reactor。

**事件调度层负责监听网络连接和读写操作，然后触发各种类型的网络事件**

## 服务编排层

服务编排层的职责是负责组装各类服务，它是 Netty 的核心处理链，用以实现网络事件的动态编排和有序传播。

服务编排层的核心组件包括 ChannelPipeline、ChannelHandler、ChannelHandlerContext。

### ChannelPipeline

ChannelPipeline 是 Netty 的核心编排组件，负责组装各种 ChannelHandler，实际数据的编解码以及加工处理操作都是由 ChannelHandler 完成的。ChannelPipeline 可以理解为ChannelHandler 的实例列表——内部通过双向链表将不同的 ChannelHandler 链接在一起。当 I/O 读写事件触发时，ChannelPipeline 会依次调用 ChannelHandler 列表对 Channel 的数据进行拦截和处理。

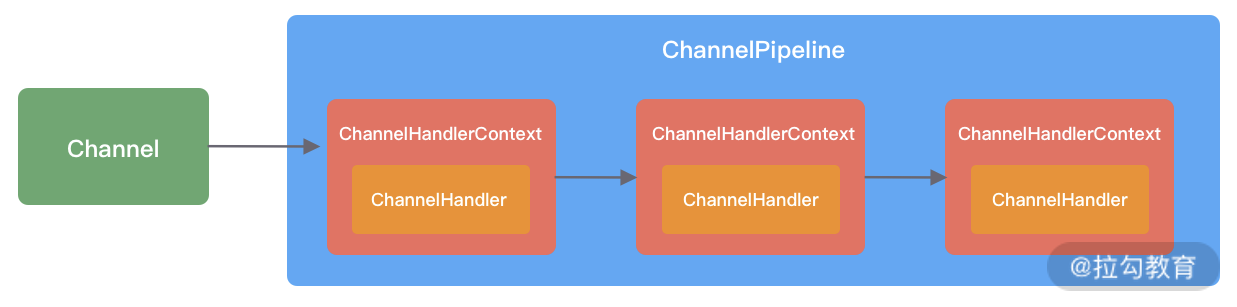
ChannelPipeline 是线程安全的，因为每一个新的 Channel 都会对应绑定一个新的 ChannelPipeline。一个 ChannelPipeline 关联一个 EventLoop，一个 EventLoop 仅会绑定一个线程。

客户端和服务端都有各自的 ChannelPipeline。以客户端为例，数据从客户端发向服务端，该过程称为出站，反之则称为入站。数据入站会由一系列 InBoundHandler 处理，然后再以相反方向的 OutBoundHandler 处理后完成出站。我们经常使用的编码 Encoder 是出站操作，解码 Decoder 是入站操作。服务端接收到客户端数据后，需要先经过 Decoder 入站处理后，再通过 Encoder 出站通知客户端。所以客户端和服务端一次完整的请求应答过程可以分为三个步骤：客户端出站（请求数据）、服务端入站（解析数据并执行业务逻辑）、服务端出站（响应结果）。

### ChannelHandler & ChannelHandlerContext

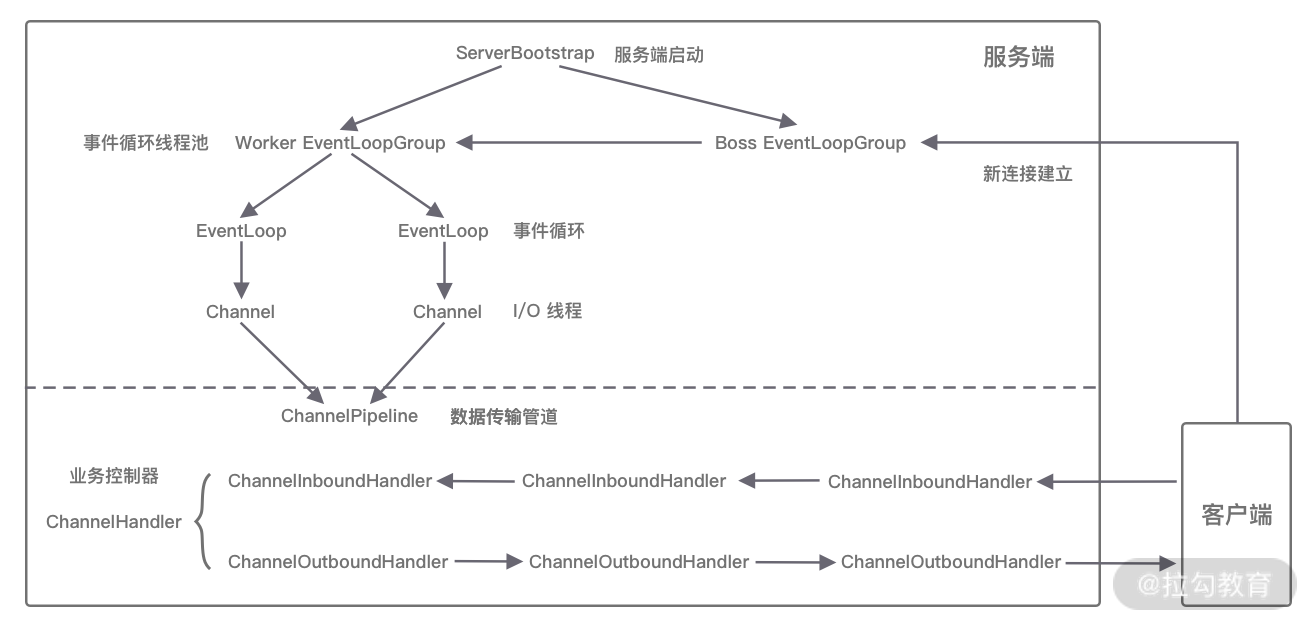
数据的编解码工作以及其他转换工作实际都是通过 ChannelHandler 处理的。

Channel 与 ChannelPipeline 的关系，从图中可以看出，每创建一个 Channel 都会绑定一个新的 ChannelPipeline，ChannelPipeline 中每加入一个 ChannelHandler 都会绑定一个 ChannelHandlerContext



ChannelHandlerContext 用于保存 ChannelHandler 上下文，通过 ChannelHandlerContext 我们可以知道 ChannelPipeline 和 ChannelHandler 的关联关系。ChannelHandlerContext 可以实现 ChannelHandler 之间的交互，ChannelHandlerContext 包含了 ChannelHandler 生命周期的所有事件，如 connect、bind、read、flush、write、close 等

## 组件关系梳理



服务端启动初始化时有 Boss EventLoopGroup 和 Worker EventLoopGroup 两个组件，其中 Boss 负责监听网络连接事件。当有新的网络连接事件到达时，则将 Channel 注册到 Worker EventLoopGroup。

Worker EventLoopGroup 会被分配一个 EventLoop 负责处理该 Channel 的读写事件。每个 EventLoop 都是单线程的，通过 Selector 进行事件循环。

当客户端发起 I/O 读写事件时，服务端 EventLoop 会进行数据的读取，然后通过 Pipeline 触发各种监听器进行数据的加工处理。

客户端数据会被传递到 ChannelPipeline 的第一个 ChannelInboundHandler 中，数据处理完成后，将加工完成的数据传递给下一个 ChannelInboundHandler。

当数据写回客户端时，会将处理结果在 ChannelPipeline 的 ChannelOutboundHandler 中传播，最后到达客户端。

# Netty服务端的启动

Netty 服务端的启动过程大致分为三个步骤：

配置线程池；

Channel 初始化；

端口绑定。

## 配置线程池

### 单线程

Reactor单线程模型所有的io操作都由一个线程完成，所以只需要启动一个eventloopgroup

EventLoopGroup group = new **NioEventLoopGroup**(1);

ServerBootstrap b = new ServerBootstrap();

b.group(group)

NioEventLoopGroup构造函数参数为1

### 多线程

Reactor 单线程模型有非常严重的性能瓶颈，因此 Reactor 多线程模型出现了。在 Netty 中使用 Reactor 多线程模型与单线程模型非常相似，区别是 NioEventLoopGroup 可以不需要任何参数，它默认会启动 2 倍 CPU 核数的线程。当然，你也可以自己手动设置固定的线程数。

EventLoopGroup group = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(group)

### 主从多线程

在大多数场景下，我们采用的都是**主从多线程 Reactor 模型**。Boss 是主 Reactor，Worker 是从 Reactor。它们分别使用不同的 NioEventLoopGroup，主 Reactor 负责处理 Accept，然后把 Channel 注册到从 Reactor 上，从 Reactor 主要负责 Channel 生命周期内的所有 I/O 事件。

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

ServerBootstrap b = new ServerBootstrap();

b.group(bossGroup, workerGroup)

## channel初始化

### 设置channel类型

NIO 模型是 Netty 中最成熟且被广泛使用的模型。因此，推荐 Netty 服务端采用 NioServerSocketChannel 作为 Channel 的类型，客户端采用 NioSocketChannel。

**b.channel(NioServerSocketChannel.class);**

Netty 提供了多种类型的 Channel 实现类，你可以按需切换，例如 OioServerSocketChannel、EpollServerSocketChannel 等。

### 主从注册channelhandler

在 Netty 中可以通过 ChannelPipeline 去注册多个 ChannelHandler，每个 ChannelHandler 各司其职，这样就可以实现最大化的代码复用，充分体现了 Netty 设计的优雅之处。

b.childHandler(new ChannelInitializer<SocketChannel>() {

    @Override

    public void initChannel(SocketChannel ch) {

        ch.pipeline()

                .addLast("codec", new HttpServerCodec())

                .addLast("compressor", new HttpContentCompressor())

                .addLast("aggregator", new HttpObjectAggregator(65536))

                .addLast("handler", new HttpServerHandler());

    }

})

ServerBootstrap 的 childHandler() 方法需要注册一个 ChannelHandler。**ChannelInitializer**是实现了 ChannelHandler**接口的匿名类**，通过实例化 ChannelInitializer 作为 ServerBootstrap 的参数。

Channel 初始化时都会绑定一个 Pipeline，它主要用于服务编排。Pipeline 管理了多个 ChannelHandler。I/O 事件依次在 ChannelHandler 中传播，

ChannelHandler 负责业务逻辑处理。上述 HTTP 服务器示例中使用链式的方式加载了多个 ChannelHandler，包含**HTTP 编解码处理器、HTTPContent 压缩处理器、HTTP 消息聚合处理器、自定义业务逻辑处理器**。

### 设置channel参数

Netty 提供了十分便捷的方法，用于设置 Channel 参数。关于 Channel 的参数数量非常多，如果每个参数都需要自己设置，那会非常繁琐。幸运的是 Netty 提供了默认参数设置，实际场景下默认参数已经满足我们的需求，我们仅需要修改自己关系的参数即可。

复制代码

1. b.option(ChannelOption.SO\_KEEPALIVE, true);

ServerBootstrap 设置 Channel 属性有**option**和**childOption**两个方法，option 主要负责设置 Boss 线程组，而 childOption 对应的是 Worker 线程组。

这里我列举了经常使用的参数含义，你可以结合业务场景，按需设置。

参数 含义

SO\_KEEPALIVE 设置为 true 代表启用了 TCP SO\_KEEPALIVE 属性，TCP 会主动探测连接状态，即连接保活

SO\_BACKLOG 已完成三次握手的请求队列最大长度，同一时刻服务端可能会处理多个连接，在高并发海量连接的场景下，该参数应适当调大

TCP\_NODELAY Netty 默认是 true，表示立即发送数据。如果设置为 false 表示启用 Nagle 算法，该算法会将 TCP 网络数据包累积到一定量才会发送，虽然可以减少报文发送的数量，但是会造成一定的数据延迟。Netty 为了最小化数据传输的延迟，默认禁用了 Nagle 算法

SO\_SNDBUF TCP 数据发送缓冲区大小

SO\_RCVBUF TCP数据接收缓冲区大小，TCP数据接收缓冲区大小

SO\_LINGER 设置延迟关闭的时间，等待缓冲区中的数据发送完成

CONNECT\_TIMEOUT\_MILLIS 建立连接的超时时间

## 绑定端口

在完成上述 Netty 的配置之后，bind() 方法会真正触发启动，sync() 方法则会阻塞，直至整个启动过程完成，具体使用方式如下：

ChannelFuture f = b.bind().sync();

# Reactor线程模型

网络框架的设计离不开 I/O 线程模型，线程模型的优劣直接决定了系统的吞吐量、可扩展性、安全性等。目前主流的网络框架几乎都采用了 I/O 多路复用的方案。

Reactor 模式作为其中的事件分发器，负责将读写事件分发给对应的读写事件处理者。

## 单线程模型

在 Reactor 单线程模型中，所有 I/O 操作（包括连接建立、数据读写、事件分发等），都是由一个线程完成的。

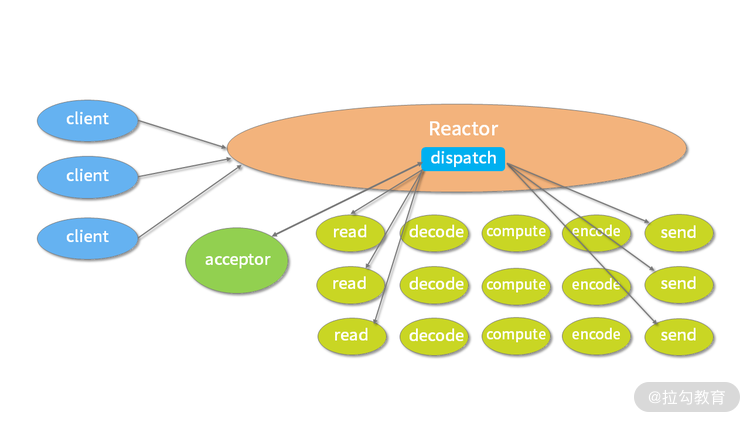
单线程模型逻辑简单，缺陷也十分明显：

一个线程支持处理的连接数非常有限，CPU 很容易打满，性能方面有明显瓶颈；

当多个事件被同时触发时，只要有一个事件没有处理完，其他后面的事件就无法执行，这就会造成消息积压及请求超时；

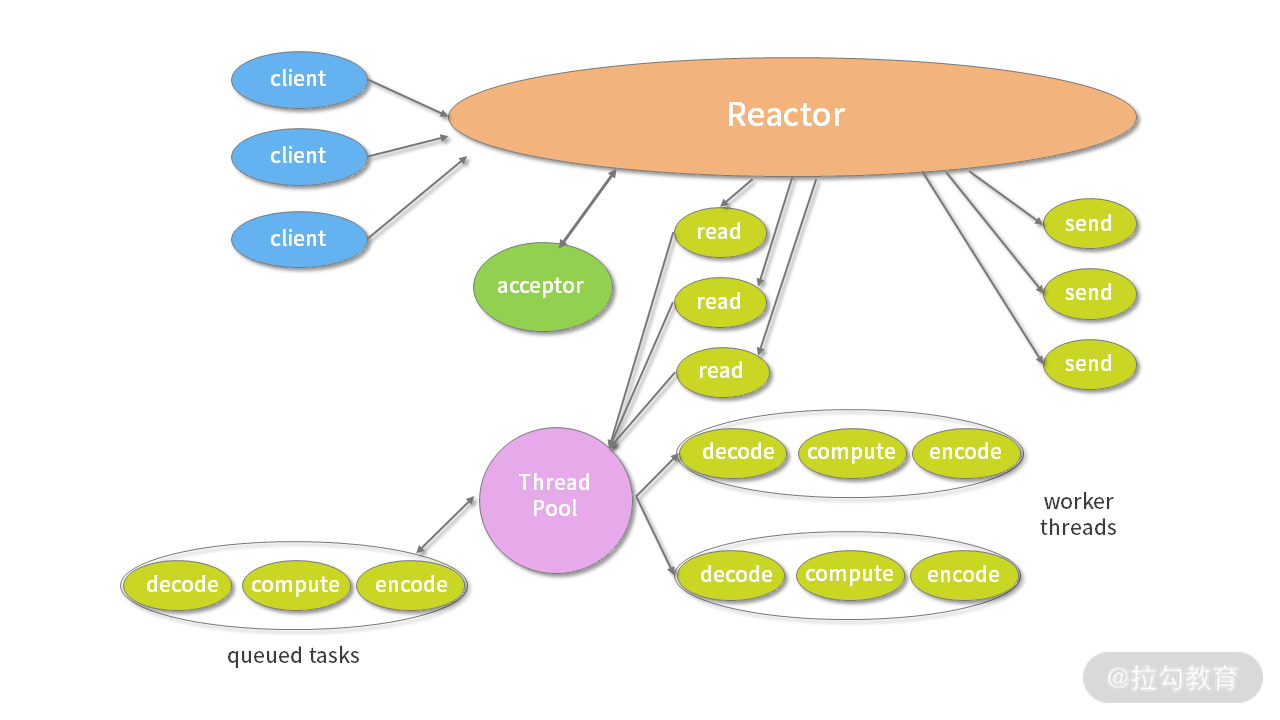
线程在处理 I/O 事件时，Select 无法同时处理连接建立、事件分发等操作；

如果 I/O 线程一直处于满负荷状态，很可能造成服务端节点不可用。



## 多线程模型

Reactor 多线程模型将业务逻辑交给多个线程进行处理。除此之外，多线程模型其他的操作与单线程模型是类似的，例如读取数据依然保留了串行化的设计。当客户端有数据发送至服务端时，Select 会监听到可读事件，数据读取完毕后提交到业务线程池中并发处理。



## 主从多线程模型

主从多线程模型由多个 Reactor 线程组成，每个 Reactor 线程都有独立的 Selector 对象。MainReactor 仅负责处理客户端连接的 Accept 事件，连接建立成功后将新创建的连接对象注册至 SubReactor。再由 SubReactor 分配线程池中的 I/O 线程与其连接绑定，它将负责连接生命周期内所有的 I/O 事件。（主线程负者接收请求数据然后封装连接对象，将封装后的对象交给从线程来分配数据）

Netty 推荐使用主从多线程模型，这样就可以轻松达到成千上万规模的客户端连接。在海量客户端并发请求的场景下，主从多线程模式甚至可以适当增加 SubReactor 线程的数量，从而利用多核能力提升系统的吞吐量。

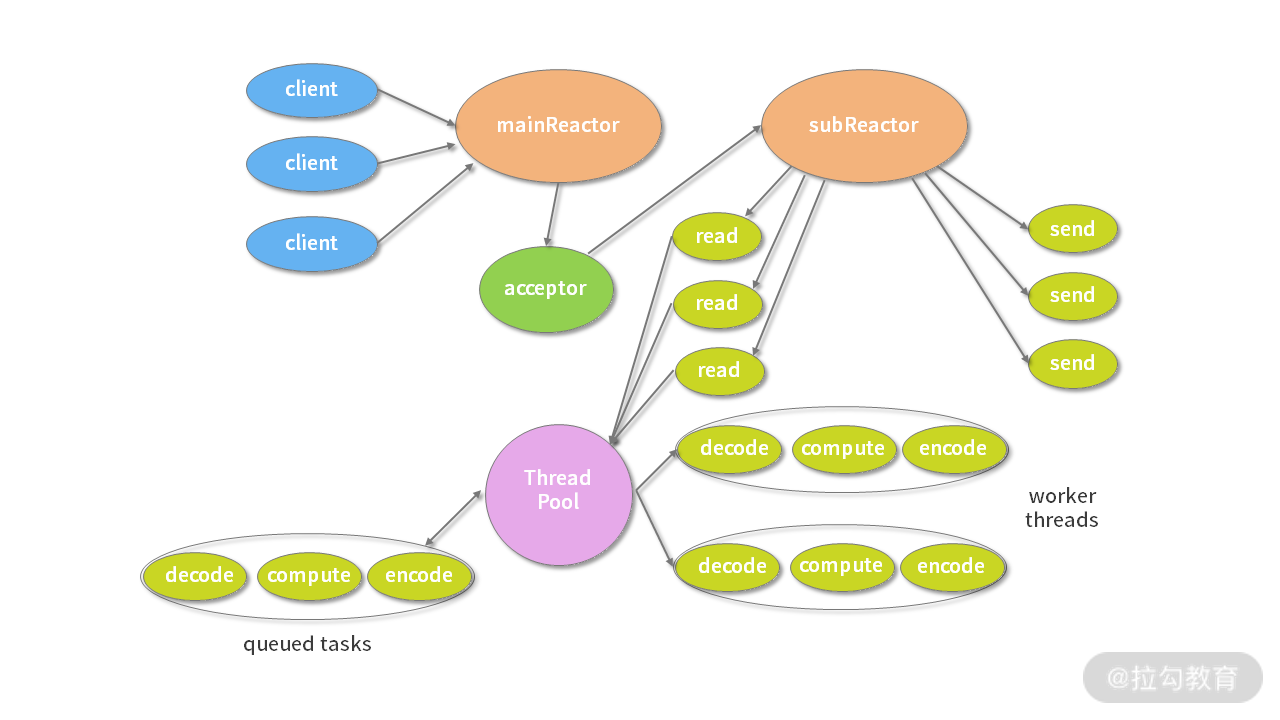
Reactor 线程模型运行机制的四个步骤，分别为连接注册、事件轮询、事件分发、任务处理

连接注册：Channel 建立后，注册至 Reactor 线程中的 Selector 选择器。

事件轮询：轮询 Selector 选择器中已注册的所有 Channel 的 I/O 事件。

事件分发：为准备就绪的 I/O 事件分配相应的处理线程。

任务处理：Reactor 线程还负责任务队列中的非 I/O 任务，每个 Worker 线程从各自维护的任务队列中取出任务异步执行。



# Netty EventLoop 实现原理

Eventloop是一种事件等待和处理的程序模型，可以解决多线程资源消耗高问题。

每当事件发生时，应用程序都会将产生的事件放入事件队列当中，然后 EventLoop 会轮询从队列中取出事件执行或者将事件分发给相应的事件监听者执行。事件执行的方式通常分为立即执行、延后执行、定期执行几种。



每个 EventLoop 线程都维护一个 Selector 选择器和任务队列 taskQueue。它主要负责处理 I/O 事件、普通任务和定时任务。

Netty 中推荐使用 NioEventLoop 作为实现类，那么 Netty 是如何实现 NioEventLoop 的呢？首先我们来看 NioEventLoop 最核心的 run() 方法源码，本节课我们不会对源码做深入的分析，只是先了解 NioEventLoop 的实现结构。

protected void run() {

    for (;;) {

        try {

            try {

                switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {

                case SelectStrategy.CONTINUE:

                    continue;

                case SelectStrategy.BUSY\_WAIT:

                case SelectStrategy.SELECT:

                    select(wakenUp.getAndSet(false)); // 轮询 I/O 事件

                    if (wakenUp.get()) {

                        selector.wakeup();

                    }

                default:

                }

            } catch (IOException e) {

                rebuildSelector0();

                handleLoopException(e);

                continue;

            }

            cancelledKeys = 0;

            needsToSelectAgain = false;

            final int ioRatio = this.ioRatio;

            if (ioRatio == 100) {

                try {

                    processSelectedKeys(); // 处理 I/O 事件

                } finally {

                    runAllTasks(); // 处理所有任务

                }

            } else {

                final long ioStartTime = System.nanoTime();

                try {

                    processSelectedKeys(); // 处理 I/O 事件

                } finally {

                    final long ioTime = System.nanoTime() - ioStartTime;

                    runAllTasks(ioTime \* (100 - ioRatio) / ioRatio); // 处理完 I/O 事件，再处理异步任务队列

                }

            }

        } catch (Throwable t) {

            handleLoopException(t);

        }

        try {

            if (isShuttingDown()) {

                closeAll();

                if (confirmShutdown()) {

                    return;

                }

            }

        } catch (Throwable t) {

            handleLoopException(t);

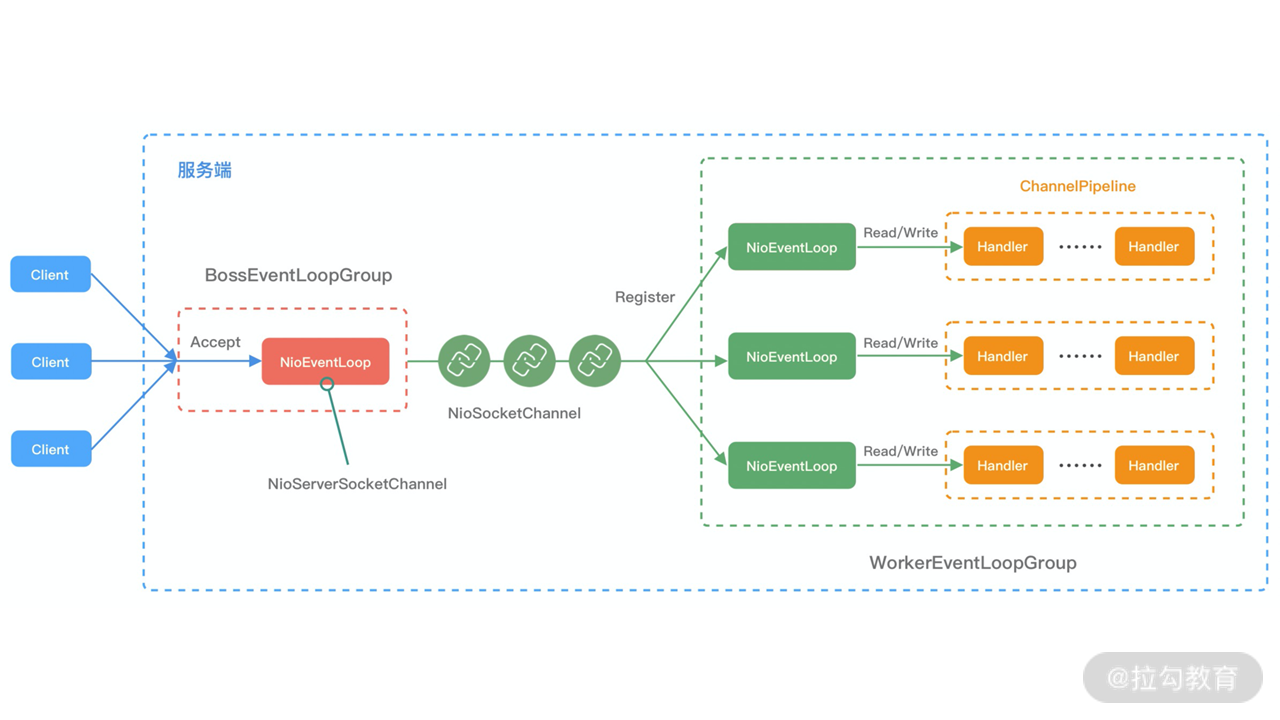
        }

    }

}

NioEventLoop 每次循环的处理流程都包含**事件轮询 select**、**事件处理 processSelectedKeys**、**任务处理 runAllTasks** 几个步骤，是典型的 Reactor 线程模型的运行机制。而且 Netty 提供了一个参数 ioRatio，可以调整 I/O 事件处理和任务处理的时间比例

## 事务处理机制



NioEventLoop 的事件处理机制采用的是无锁串行化的设计思路

BossEventLoopGroup 和 WorkerEventLoopGroup 包含一个或者多个 NioEventLoop。BossEventLoopGroup 负责监听客户端的 Accept 事件，当事件触发时，将事件注册至 WorkerEventLoopGroup 中的一个 NioEventLoop 上。每新建一个 Channel， 只选择一个 NioEventLoop 与其绑定。所以说 Channel 生命周期的所有事件处理都是线程独立的，不同的 NioEventLoop 线程之间不会发生任何交集。

NioEventLoop 完成数据读取后，会调用绑定的 ChannelPipeline 进行事件传播，ChannelPipeline 也是线程安全的，数据会被传递到 ChannelPipeline 的第一个 ChannelHandler 中。数据处理完成后，将加工完成的数据再传递给下一个 ChannelHandler，整个过程是串行化执行，不会发生线程上下文切换的问题。

NioEventLoop 无锁串行化的设计不仅使系统吞吐量达到最大化，而且降低了用户开发业务逻辑的难度，不需要花太多精力关心线程安全问题。虽然单线程执行避免了线程切换，但是它的缺陷就是**不能执行时间过长的 I/O 操作，一旦某个 I/O 事件发生阻塞，那么后续的所有 I/O 事件都无法执行，甚至造成事件积压**。在使用 Netty 进行程序开发时，我们一定要对 ChannelHandler 的实现逻辑有充分的风险意识。

NioEventLoop 线程的可靠性至关重要，一旦 NioEventLoop 发生阻塞或者陷入空轮询，就会导致整个系统不可用。在 JDK 中， Epoll 的实现是存在漏洞的，即使 Selector 轮询的事件列表为空，NIO 线程一样可以被唤醒，导致 CPU 100% 占用。这就是臭名昭著的 **JDK epoll 空轮询的 Bug**。

long time = System.nanoTime();

if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos) {

    selectCnt = 1;

} else if (SELECTOR\_AUTO\_REBUILD\_THRESHOLD > 0 &&

        selectCnt >= SELECTOR\_AUTO\_REBUILD\_THRESHOLD) {

    selector = selectRebuildSelector(selectCnt);

    selectCnt = 1;

    break;

}

Netty 提供了一种检测机制判断线程是否可能陷入空轮询，具体的实现方式如下：

每次执行 Select 操作之前记录当前时间 currentTimeNanos。

time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos，如果事件轮询的持续时间大于等于 timeoutMillis，那么说明是正常的，否则表明阻塞时间并未达到预期，可能触发了空轮询的 Bug。

Netty 引入了计数变量 selectCnt。在正常情况下，selectCnt 会重置，否则会对 selectCnt 自增计数。当 selectCnt 达到 SELECTOR\_AUTO\_REBUILD\_THRESHOLD（默认512） 阈值时，会触发重建 Selector 对象。

## 任务处理机制

NioEventLoop 不仅负责处理 I/O 事件，还要兼顾执行任务队列中的任务。任务队列遵循 FIFO 规则，可以保证任务执行的公平性。NioEventLoop 处理的任务类型基本可以分为三类。

普通任务：通过 NioEventLoop 的 execute() 方法向任务队列 taskQueue 中添加任务。例如 Netty 在写数据时会封装 WriteAndFlushTask 提交给 taskQueue。taskQueue 的实现类是多生产者单消费者队列 MpscChunkedArrayQueue，在多线程并发添加任务时，可以保证线程安全。

定时任务：通过调用 NioEventLoop 的 schedule() 方法向定时任务队列 scheduledTaskQueue 添加一个定时任务，用于周期性执行该任务。例如，心跳消息发送等。定时任务队列 scheduledTaskQueue 采用优先队列 PriorityQueue 实现。

尾部队列：tailTasks 相比于普通任务队列优先级较低，在每次执行完 taskQueue 中任务后会去获取尾部队列中任务执行。尾部任务并不常用，主要用于做一些收尾工作，例如统计事件循环的执行时间、监控信息上报等。

## EventLoop 最佳实践

在日常开发中用好 EventLoop 至关重要，这里结合实际工作中的经验给出一些 EventLoop 的最佳实践方案。

网络连接建立过程中三次握手、安全认证的过程会消耗不少时间。这里建议采用 Boss 和 Worker 两个 EventLoopGroup，有助于分担 Reactor 线程的压力。

由于 Reactor 线程模式适合处理耗时短的任务场景，对于耗时较长的 ChannelHandler 可以考虑维护一个业务线程池，将编解码后的数据封装成 Task 进行异步处理，避免 ChannelHandler 阻塞而造成 EventLoop 不可用。

如果业务逻辑执行时间较短，建议直接在 ChannelHandler 中执行。例如编解码操作，这样可以避免过度设计而造成架构的复杂性。

不宜设计过多的 ChannelHandler。对于系统性能和可维护性都会存在问题，在设计业务架构的时候，需要明确业务分层和 Netty 分层之间的界限。不要一味地将业务逻辑都添加到 ChannelHandler 中。

**Netty EventLoop 的功能用处做一个简单的归纳总结。**

MainReactor 线程：处理客户端请求接入。

SubReactor 线程：数据读取、I/O 事件的分发与执行。

任务处理线程：用于执行普通任务或者定时任务，如空闲连接检测、心跳上报等。

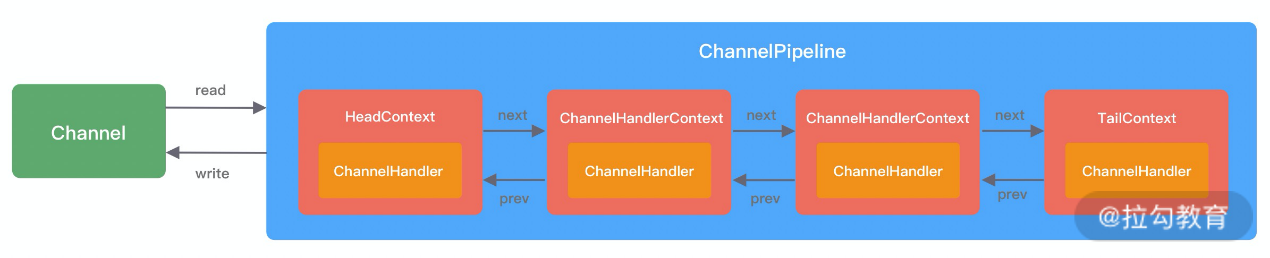
# ChannelPipeline 概述

EventLoop 可以说是 Netty 的调度中心，负责监听多种事件类型：I/O 事件、信号事件、定时事件等，然而实际的业务处理逻辑则是由 ChannelPipeline 中所定义的 ChannelHandler 完成的

## ChannelPipeline 内部结构

ChannelPipeline 作为 Netty 的核心编排组件，负责调度各种类型的 ChannelHandler，实际数据的加工处理操作则是由 ChannelHandler 完成的。

ChannelPipeline 可以看作是 ChannelHandler 的容器载体，它是由一组 ChannelHandler 实例组成的，内部通过双向链表将不同的 ChannelHandler 链接在一起. 当有 I/O 读写事件触发时，ChannelPipeline 会依次调用 ChannelHandler 列表对 Channel 的数据进行拦截和处理。



每个 Channel 会绑定一个 ChannelPipeline，每一个 ChannelPipeline 都包含多个 ChannelHandlerContext，所有 ChannelHandlerContext 之间组成了双向链表。又因为每个 ChannelHandler 都对应一个 ChannelHandlerContext，所以实际上 ChannelPipeline 维护的是它与 ChannelHandlerContext 的关系.

ChannelHandlerContext 则包含了 ChannelHandler 生命周期的所有事件，如 connect、bind、read、flush、write、close 等。如果没有 ChannelHandlerContext 的这层封装，那么我们在做 ChannelHandler 之间传递的时候，前置后置的通用逻辑就要在每个 ChannelHandler 里都实现一份。这样虽然能解决问题，但是代码结构的耦合，会非常不优雅。

ChannelPipeline 的双向链表分别维护了 HeadContext 和 TailContext 的头尾节点。我们自定义的 ChannelHandler 会插入到 Head 和 Tail 之间，这两个节点在 Netty 中已经默认实现了，它们在 ChannelPipeline 中起到了至关重要的作用

HeadContext 既是 Inbound 处理器，也是 Outbound 处理器。它分别实现了 ChannelInboundHandler 和 ChannelOutboundHandler。网络数据写入操作的入口就是由 HeadContext 节点完成的。HeadContext 作为 Pipeline 的头结点负责读取数据并开始传递 InBound 事件，当数据处理完成后，数据会反方向经过 Outbound 处理器，最终传递到 HeadContext，所以 HeadContext 又是处理 Outbound 事件的最后一站。

TailContext 只实现了 ChannelInboundHandler 接口。它会在 ChannelInboundHandler 调用链路的最后一步执行，主要用于终止 Inbound 事件传播，例如释放 Message 数据资源等。TailContext 节点作为 OutBound 事件传播的第一站，仅仅是将 OutBound 事件传递给上一个节点。

## ChannelHandler 接口设计

整个 ChannelHandler 是围绕 I/O 事件的生命周期所设计的，例如建立连接、读数据、写数据、连接销毁等。ChannelHandler 有两个重要的子接口：ChannelInboundHandler和ChannelOutboundHandler，分别拦截入站和出站的各种 I/O 事件。

1. ChannelInboundHandler 的事件回调方法与触发时机。

事件回调方法 触发时机

channelRegistered Channel 被注册到 EventLoop

channelUnregistered Channel 从 EventLoop 中取消注册

channelActive Channel 处于就绪状态，可以被读写

channelInactive Channel 处于非就绪状态Channel 可以从远端读取到数据

channelRead Channel 可以从远端读取到数据

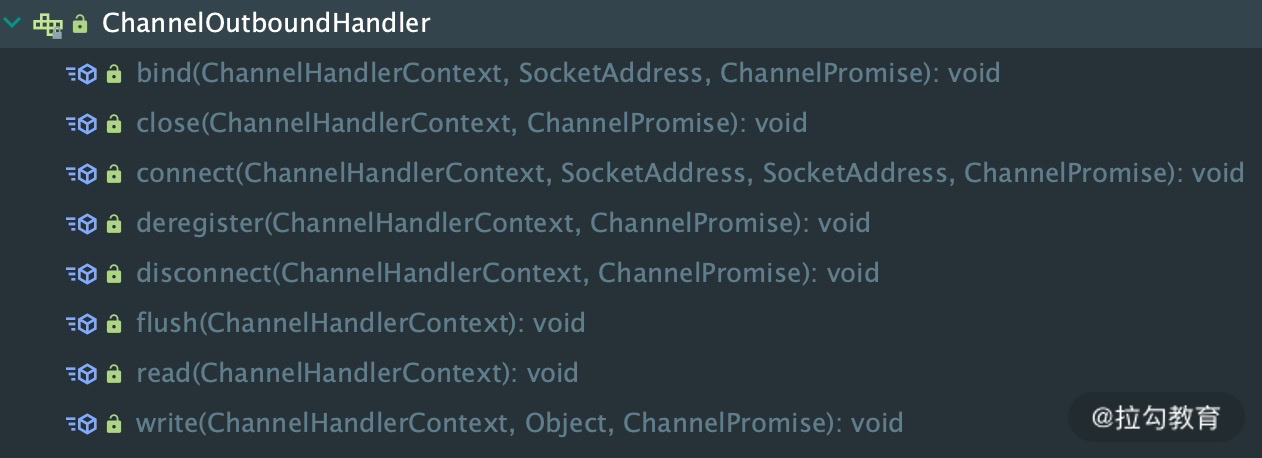
channelReadComplete Channel 读取数据完成

userEventTriggered 用户事件触发时

channelWritabilityChanged Channel 的写状态发生变化

2. ChannelOutboundHandler 的事件回调方法与触发时机。

ChannelOutboundHandler 的事件回调方法非常清晰，直接通过 ChannelOutboundHandler 的接口列表可以看到每种操作所对应的回调方法



## 事件传播机制

ChannelPipeline 可分为入站 ChannelInboundHandler 和出站 ChannelOutboundHandler 两种处理器，与此对应传输的事件类型可以分为**Inbound 事件**和**Outbound 事件**。

serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {

    @Override

    public void initChannel(SocketChannel ch) {

        ch.pipeline()

                .addLast(new SampleInBoundHandler("SampleInBoundHandlerA", false))

                .addLast(new SampleInBoundHandler("SampleInBoundHandlerB", false))

                .addLast(new SampleInBoundHandler("SampleInBoundHandlerC", true));

        ch.pipeline()

                .addLast(new SampleOutBoundHandler("SampleOutBoundHandlerA"))

                .addLast(new SampleOutBoundHandler("SampleOutBoundHandlerB"))

                .addLast(new SampleOutBoundHandler("SampleOutBoundHandlerC"));

    }

}

public class SampleInBoundHandler extends ChannelInboundHandlerAdapter {

    private final String name;

    private final boolean flush;

    public SampleInBoundHandler(String name, boolean flush) {

        this.name = name;

        this.flush = flush;

    }

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

        System.out.println("InBoundHandler: " + name);

        if (flush) {

            ctx.channel().writeAndFlush(msg);

        } else {

            super.channelRead(ctx, msg);

        }

    }

}

public class SampleOutBoundHandler extends ChannelOutboundHandlerAdapter {

    private final String name;

    public SampleOutBoundHandler(String name) {

        this.name = name;

    }

    @Override

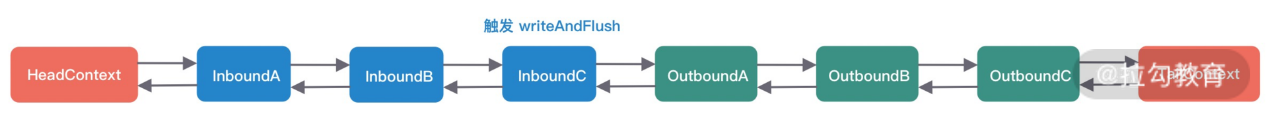
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {

        System.out.println("OutBoundHandler: " + name);

        super.write(ctx, msg, promise);

    }

}

通过 Pipeline 的 addLast 方法分别添加了三个 InboundHandler 和 OutboundHandler，添加顺序都是 A -> B -> C，下图可以表示初始化后 ChannelPipeline 的内部结构。

当客户端向服务端发送请求时，会触发 SampleInBoundHandler 调用链的 channelRead 事件。经过 SampleInBoundHandler 调用链处理完成后，在 SampleInBoundHandlerC 中会调用 writeAndFlush 方法向客户端写回数据，此时会触发 SampleOutBoundHandler 调用链的 write 事件。

Inbound 事件和 Outbound 事件的传播方向是不一样的。Inbound 事件的传播方向为 Head -> Tail，而 Outbound 事件传播方向是 Tail -> Head

## 异常传播机制

ChannelPipeline 事件传播的实现采用了经典的责任链模式，调用链路环环相扣;

那么如果有一个节点处理逻辑异常会出现什么现象呢？我们通过修改 SampleInBoundHandler 的实现来模拟业务逻辑异常：

public class SampleInBoundHandler extends ChannelInboundHandlerAdapter {

    private final String name;

    private final boolean flush;

    public SampleInBoundHandler(String name, boolean flush) {

        this.name = name;

        this.flush = flush;

    }

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg) {

        System.out.println("InBoundHandler: " + name);

        if (flush) {

            ctx.channel().writeAndFlush(msg);

        } else {

            throw new RuntimeException("InBoundHandler: " + name);

        }

    }

    @Override

    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

        System.out.println("InBoundHandlerException: " + name);

        ctx.fireExceptionCaught(cause);

    }

}

在 channelRead 事件处理中，第一个 A 节点就会抛出 RuntimeException。同时我们重写了 ChannelInboundHandlerAdapter 中的 exceptionCaught 方法, 只是在开头加上了控制台输出，方便观察异常传播的行为。

由输出结果可以看出 ctx.fireExceptionCaugh 会将异常按顺序从 Head 节点传播到 Tail 节点。如果用户没有对异常进行拦截处理，最后将由 Tail 节点统一处理

### 异常处理的最佳实践

最好的方法是在 ChannelPipeline 自定义处理器的末端添加统一的异常处理器



public class ExceptionHandler extends ChannelDuplexHandler {

    @Override

    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {

        if (cause instanceof RuntimeException) {

            System.out.println("Handle Business Exception Success.");

        }

    }

}

加入统一的异常处理器后，可以看到异常已经被优雅地拦截并处理掉了。这也是 Netty 推荐的最佳异常处理实践。

# Netty实现自定义通信协议

所谓协议，就是通信双方事先商量好的接口暗语，在 TCP 网络编程中，发送方和接收方的数据包格式都是二进制，发送方将对象转化成二进制流发送给接收方，接收方获得二进制数据后需要知道如何解析成对象，所以协议是双方能够正常通信的基础。

目前市面上已经有不少通用的协议，例如 HTTP、HTTPS、JSON-RPC、FTP、IMAP、Protobuf 等。通用协议兼容性好，易于维护，各种异构系统之间可以实现无缝对接。

如果在满足业务场景以及性能需求的前提下，推荐采用通用协议的方案。相比通用协议，自定义协议主要有以下优点。

**极致性能**：通用的通信协议考虑了很多兼容性的因素，必然在性能方面有所损失。

**扩展性**：自定义的协议相比通用协议更好扩展，可以更好地满足自己的业务需求。

**安全性**：通用协议是公开的，很多漏洞已经很多被黑客攻破。自定义协议更加安全，因为黑客需要先破解你的协议内容。

## 网络协议需要具备的要素

### 魔数

魔数是通信双方协商的一个暗号，通常采用固定的几个字节表示。魔数的作用是防止任何人随便向服务器的端口上发送数据。

### 协议版本号

随着业务需求的变化，协议可能需要对结构或字段进行改动，不同版本的协议对应的解析方法也是不同的。所以在生产级项目中强烈建议预留协议版本号这个字段。

### 序列化算法

序列化算法字段表示数据发送方应该采用何种方法将请求的对象转化为二进制，以及如何再将二进制转化为对象，如 JSON、Hessian、Java 自带序列化等。

### 报文类型

在不同的业务场景中，报文可能存在不同的类型。例如在 RPC 框架中有请求、响应、心跳等类型的报文，在 IM 即时通信的场景中有登陆、创建群聊、发送消息、接收消息、退出群聊等类型的报文。

### 长度域字段

长度域字段代表请求数据的长度，接收方根据长度域字段获取一个完整的报文。

### 请求数据

请求数据通常为序列化之后得到的二进制流，每种请求数据的内容是不一样的。

### 状态

状态字段用于标识请求是否正常。一般由被调用方设置。例如一次 RPC 调用失败，状态字段可被服务提供方设置为异常状态。

### 保留字段

保留字段是可选项，为了应对协议升级的可能性，可以预留若干字节的保留字段，以备不时之需。

## Netty 如何实现自定义通信协议

Netty 作为一个非常优秀的网络通信框架，已经为我们提供了非常丰富的编解码抽象基类，帮助我们更方便地基于这些抽象基类扩展实现自定义协议。

### Netty常用的编码器

MessageToByteEncoder 对象编码成字节流；

MessageToMessageEncoder 一种消息类型编码成另外一种消息类型。

### Netty 常用解码器

ByteToMessageDecoder/ReplayingDecoder 将字节流解码为消息对象；

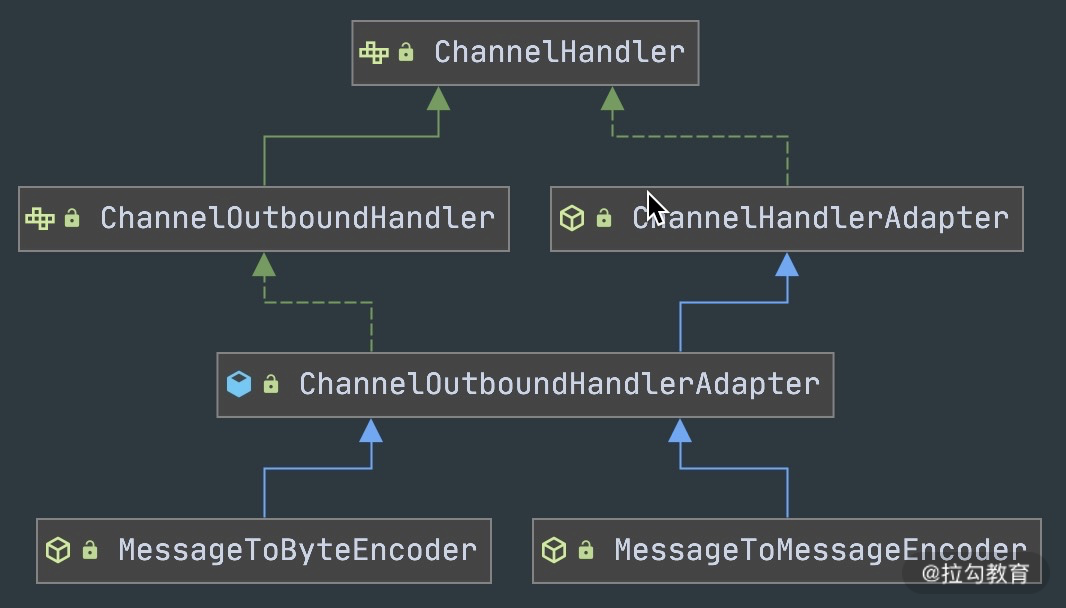
MessageToMessageDecoder 将一种消息类型解码为另外一种消息类型。

编解码器可以分为一次解码器和二次解码器，一次解码器用于解决 TCP 拆包/粘包问题，按协议解析后得到的字节数据。如果你需要对解析后的字节数据做对象模型的转换，这时候便需要用到二次解码器，同理编码器的过程是反过来的。

一次编解码器：MessageToByteEncoder/ByteToMessageDecoder。

二次编解码器：MessageToMessageEncoder/MessageToMessageDecoder。

### 抽象编码类



MessageToByteEncoder

MessageToByteEncoder 用于将对象编码成字节流，MessageToByteEncoder 提供了唯一的 encode 抽象方法，我们只需要实现encode 方法即可完成自定义编码。那么encode() 方法是在什么时候被调用的呢？我们一起看下MessageToByteEncoder 的核心源码片段，如下所示。

@Override

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {

ByteBuf buf = null;

try {

if (acceptOutboundMessage(msg)) { // 1. 消息类型是否匹配

@SuppressWarnings("unchecked")

I cast = (I) msg;

buf = allocateBuffer(ctx, cast, preferDirect); // 2. 分配 ByteBuf 资源

try {

encode(ctx, cast, buf); // 3. 执行 encode 方法完成数据编码

} finally {

ReferenceCountUtil.release(cast);

}

if (buf.isReadable()) {

ctx.write(buf, promise); // 4. 向后传递写事件

} else {

buf.release();

ctx.write(Unpooled.EMPTY\_BUFFER, promise);

}

buf = null;

} else {

ctx.write(msg, promise);

}

} catch (EncoderException e) {

throw e;

} catch (Throwable e) {

throw new EncoderException(e);

} finally {

if (buf != null) {

buf.release();

}

}

}

MessageToByteEncoder 重写了 ChanneOutboundHandler 的 write() 方法，其主要逻辑分为以下几个步骤：

1. acceptOutboundMessage 判断是否有匹配的消息类型，如果匹配需要执行编码流程，如果不匹配直接继续传递给下一个 ChannelOutboundHandler；
2. 分配 ByteBuf 资源，默认使用**堆外内存；**
3. 调用子类实现的 encode 方法完成数据编码，一旦消息被成功编码，会通过调用 ReferenceCountUtil.release(cast) 自动释放；
4. 如果 ByteBuf 可读，说明已经成功编码得到数据，然后写入 ChannelHandlerContext 交到下一个节点；如果 ByteBuf 不可读，则释放 ByteBuf 资源，向下传递空的 ByteBuf 对象。

编码器实现非常简单，不需要关注拆包/粘包问题。如下例子，展示了如何将字符串类型的数据写入到 ByteBuf 实例，ByteBuf 实例将传递给 ChannelPipeline 链表中的下一个 ChannelOutboundHandler。

public class StringToByteEncoder extends MessageToByteEncoder<String> {

        @Override

        protected void encode(ChannelHandlerContext channelHandlerContext, String data, ByteBuf byteBuf) throws Exception {

            byteBuf.writeBytes(data.getBytes());

        }

}

**MessageToMessageEncoder**

MessageToMessageEncoder 与 MessageToByteEncoder 类似，同样只需要实现 encode 方法。与 MessageToByteEncoder 不同的是，MessageToMessageEncoder 是将一种格式的消息转换为另外一种格式的消息。其中第二个 Message 所指的可以是任意一个对象，如果该对象是 ByteBuf 类型，那么基本上和 MessageToByteEncoder 的实现原理是一致的。此外 MessageToByteEncoder 的输出结果是对象列表，编码后的结果属于**中间对象**，最终仍然会转化成 ByteBuf 进行传输。

MessageToMessageEncoder 常用的**实现子类**有 StringEncoder、LineEncoder、Base64Encoder 等。以 StringEncoder 为例看下 MessageToMessageEncoder 的用法。源码示例如下：将 CharSequence 类型（String、StringBuilder、StringBuffer 等）转换成 ByteBuf 类型，结合 StringDecoder 可以直接实现 String 类型数据的编解码。

@Override

protected void encode(ChannelHandlerContext ctx, CharSequence msg, List<Object> out) throws Exception {

    if (msg.length() == 0) {

        return;

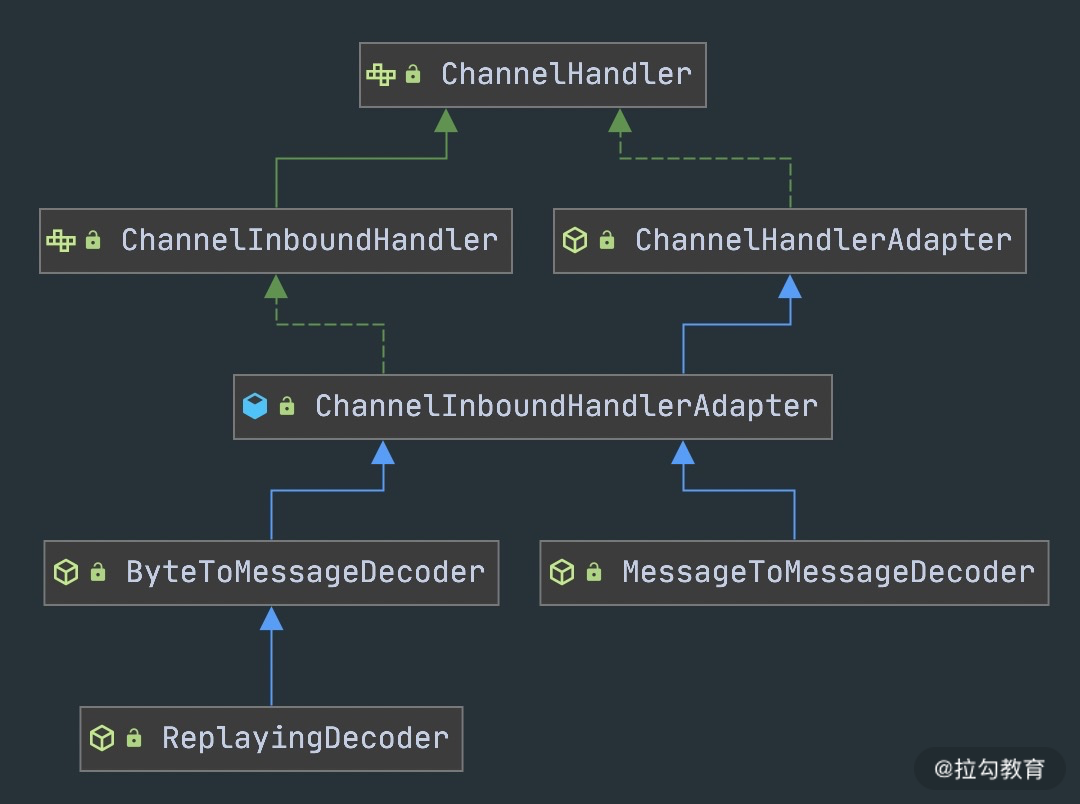
    }

    out.add(ByteBufUtil.encodeString(ctx.alloc(), CharBuffer.wrap(msg), charset));

}

### 抽象解码类

同样，我们先看下抽象解码类的继承关系图。解码类是 ChanneInboundHandler 的抽象类实现，操作的是 Inbound 入站数据。解码器实现的难度要远大于编码器，因为解码器需要考虑拆包/粘包问题。由于接收方有可能没有接收到完整的消息，所以解码框架需要对入站的数据做缓冲操作，直至获取到完整的消息。



**抽象解码类 ByteToMessageDecoder。**

首先，我们看下 ByteToMessageDecoder 定义的抽象方法：

public abstract class ByteToMessageDecoder extends ChannelInboundHandlerAdapter {

    protected abstract void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception;

    protected void decodeLast(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {

        if (in.isReadable()) {

            decodeRemovalReentryProtection(ctx, in, out);

        }

    }

}

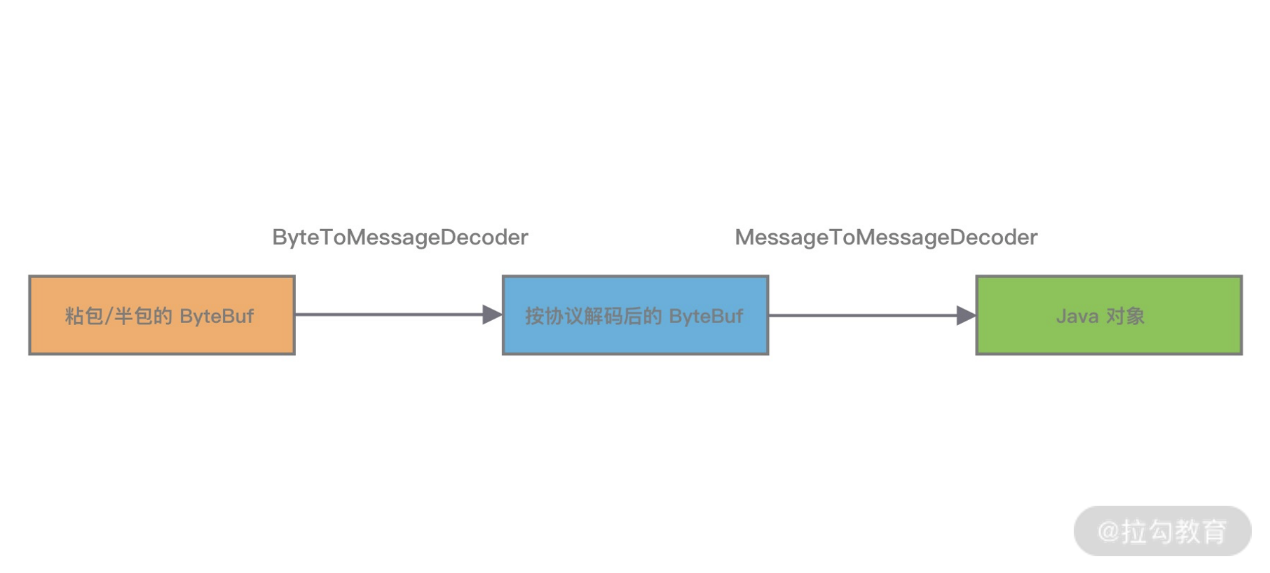
decode() 是用户必须实现的抽象方法，在该方法在调用时需要传入接收的数据 ByteBuf，及用来添加编码后消息的 List。**由于 TCP 粘包问题，ByteBuf 中可能包含多个有效的报文，或者不够一个完整的报文。Netty 会重复回调 decode() 方法，直到没有解码出新的完整报文可以添加到 List 当中，或者 ByteBuf 没有更多可读取的数据为止**。如果此时 List 的内容不为空，那么会传递给  ChannelPipeline 中的下一个ChannelInboundHandler。

此外 ByteToMessageDecoder 还定义了 decodeLast() 方法。为什么抽象解码器要比编码器多一个 decodeLast() 方法呢？因为 decodeLast 在 Channel 关闭后会被调用一次，主要用于处理 ByteBuf 最后剩余的字节数据。Netty 中 decodeLast 的默认实现只是简单调用了 decode() 方法。如果有特殊的业务需求，则可以通过重写 decodeLast() 方法扩展自定义逻辑。

ByteToMessageDecoder 还有一个抽象子类是 ReplayingDecoder。它封装了缓冲区的管理，在读取缓冲区数据时，你无须再对字节长度进行检查。因为如果没有足够长度的字节数据，ReplayingDecoder 将终止解码操作。ReplayingDecoder 的性能相比直接使用 ByteToMessageDecoder 要慢，大部分情况下并不推荐使用 ReplayingDecoder。

**抽象解码类 MessageToMessageDecoder。**

MessageToMessageDecoder 与 ByteToMessageDecoder 作用类似，都是将一种消息类型的编码成另外一种消息类型。与 ByteToMessageDecoder 不同的是 MessageToMessageDecoder 并不会对数据报文进行缓存，它主要用作转换消息模型。比较推荐的做法是使用 ByteToMessageDecoder 解析 TCP 协议，解决拆包/粘包问题。解析得到有效的 ByteBuf 数据，然后传递给后续的 MessageToMessageDecoder 做数据对象的转换，具体流程如下图所示。



### 通信协议实战

实现协议编码器之前，我们首先需要清楚一个问题：如何判断 ByteBuf 是否存在完整的报文？最常用的做法就是通过读取消息长度 dataLength 进行判断。如果 ByteBuf 的可读数据长度小于 dataLength，说明 ByteBuf 还不够获取一个完整的报文。在该协议前面的消息头部分包含了魔数、协议版本号、数据长度等固定字段，共 14 个字节。固定字段长度和数据长度可以作为我们判断消息完整性的依据，具体编码器实现逻辑示例如下：

/\*

+---------------------------------------------------------------+

| 魔数 2byte | 协议版本号 1byte | 序列化算法 1byte | 报文类型 1byte  |

+---------------------------------------------------------------+

| 状态 1byte |        保留字段 4byte     |      数据长度 4byte     |

+---------------------------------------------------------------+

|                   数据内容 （长度不定）                          |

+---------------------------------------------------------------+

 \*/

@Override

public final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {

    // 判断 ByteBuf 可读取字节

    if (in.readableBytes() < 14) {

        return;

    }

    in.markReaderIndex(); // 标记 ByteBuf 读指针位置

    in.skipBytes(2); // 跳过魔数

    in.skipBytes(1); // 跳过协议版本号

    byte serializeType = in.readByte();

    in.skipBytes(1); // 跳过报文类型

    in.skipBytes(1); // 跳过状态字段

    in.skipBytes(4); // 跳过保留字段

    int dataLength = in.readInt();

    if (in.readableBytes() < dataLength) {

        in.resetReaderIndex(); // 重置 ByteBuf 读指针位置

        return;

    }

    byte[] data = new byte[dataLength];

    in.readBytes(data);

    SerializeService serializeService = getSerializeServiceByType(serializeType);

    Object obj = serializeService.deserialize(data);

    if (obj != null) {

        out.add(obj);

    }

}

# Netty常用的解码器

## 固定长度解码器 FixedLengthFrameDecoder

固定长度解码器 FixedLengthFrameDecoder 非常简单，直接通过构造函数设置固定长度的大小 frameLength，无论接收方一次获取多大的数据，都会严格按照 frameLength 进行解码。如果累积读取到长度大小为 frameLength 的消息，那么解码器认为已经获取到了一个完整的消息。如果消息长度小于 frameLength，FixedLengthFrameDecoder 解码器会一直等后续数据包的到达，直至获得完整的消息。

public class EchoServer {

    public void startEchoServer(int port) throws Exception {

        EventLoopGroup bossGroup = new NioEventLoopGroup();

        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {

            ServerBootstrap b = new ServerBootstrap();

            b.group(bossGroup, workerGroup)

                    .channel(NioServerSocketChannel.class)

                    .childHandler(new ChannelInitializer<SocketChannel>() {

                        @Override

                        public void initChannel(SocketChannel ch) {

                            ch.pipeline().addLast(new FixedLengthFrameDecoder(10));

                            ch.pipeline().addLast(new EchoServerHandler());

                        }

                    });

            ChannelFuture f = b.bind(port).sync();

            f.channel().closeFuture().sync();

        } finally {

            bossGroup.shutdownGracefully();

            workerGroup.shutdownGracefully();

        }

    }

    public static void main(String[] args) throws Exception {

        new EchoServer().startEchoServer(8088);

    }

}

@Sharable

public class EchoServerHandler extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg) {

        System.out.println("Receive client : [" + ((ByteBuf) msg).toString(CharsetUtil.UTF\_8) + "]");

    }

}

## 特殊分隔符解码器 DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder的常用的属性

**Delimiters**

delimiters 指定特殊分隔符，通过写入 ByteBuf 作为**参数**传入。delimiters 的类型是 ByteBuf 数组，所以我们可以同时指定多个分隔符，但是最终会选择长度最短的分隔符进行消息拆分。

**maxLength**

maxLength 是报文最大长度的限制。如果超过 maxLength 还没有检测到指定分隔符，将会抛出 TooLongFrameException。可以说 maxLength 是对程序在极端情况下的一种**保护措施**。

**failFast**

failFast 与 maxLength 需要搭配使用，通过设置 failFast 可以控制抛出 TooLongFrameException 的时机，可以说 Netty 在细节上考虑得面面俱到。如果 failFast=true，那么在超出 maxLength 会立即抛出 TooLongFrameException，不再继续进行解码。如果 failFast=false，那么会等到解码出一个完整的消息后才会抛出 TooLongFrameException。

**stripDelimiter**

stripDelimiter 的作用是判断解码后得到的消息是否去除分隔符

## 长度域解码器 LengthFieldBasedFrameDecoder

长度域解码器 LengthFieldBasedFrameDecoder 是解决 TCP 拆包/粘包问题最常用的\*\*解码器。\*\*它基本上可以覆盖大部分基于长度拆包场景，开源消息中间件 RocketMQ 就是使用 LengthFieldBasedFrameDecoder 进行解码的。LengthFieldBasedFrameDecoder 相比 FixedLengthFrameDecoder 和 DelimiterBasedFrameDecoder 要复杂一些，接下来我们就一起学习下这个强大的解码器。

# 数据传输：writeAndFlush 处理流程剖析

在 Netty 中实现数据发送非常简单，只需要调用 writeAndFlush 方法即可

## writeAndFlush 事件传播分析

以下是服务端的启动类，分别注册了三个 ChannelHandler：**固定长度解码器 FixedLengthFrameDecoder**、**响应结果编码器 ResponseSampleEncoder**、**业务逻辑处理器 RequestSampleHandler**。

public class EchoServer {

    public void startEchoServer(int port) throws Exception {

        EventLoopGroup bossGroup = new NioEventLoopGroup();

        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {

            ServerBootstrap b = new ServerBootstrap();

            b.group(bossGroup, workerGroup)

                    .channel(NioServerSocketChannel.class)

                    .childHandler(new ChannelInitializer<SocketChannel>() {

                        @Override

                        public void initChannel(SocketChannel ch) {

                            ch.pipeline().addLast(new FixedLengthFrameDecoder(10));

                            ch.pipeline().addLast(new ResponseSampleEncoder());

                            ch.pipeline().addLast(new RequestSampleHandler());

                        }

                    });

            ChannelFuture f = b.bind(port).sync();

            f.channel().closeFuture().sync();

        } finally {

            bossGroup.shutdownGracefully();

            workerGroup.shutdownGracefully();

        }

    }

    public static void main(String[] args) throws Exception {

        new EchoServer().startEchoServer(8088);

    }

}

其中固定长度解码器 FixedLengthFrameDecoder 是 Netty 自带的解码器，在这里就不做赘述了。下面我们分别看下另外两个 ChannelHandler 的具体实现。

响应结果编码器 ResponseSampleEncoder 用于将服务端的处理结果进行编码，具体的实现逻辑如下：

public class ResponseSampleEncoder extends MessageToByteEncoder<ResponseSample> {

    @Override

    protected void encode(ChannelHandlerContext ctx, ResponseSample msg, ByteBuf out) {

        if (msg != null) {

            out.writeBytes(msg.getCode().getBytes());

            out.writeBytes(msg.getData().getBytes());

            out.writeLong(msg.getTimestamp());

        }

    }

}

RequestSampleHandler 主要负责客户端的数据处理，并通过调用 ctx.channel().writeAndFlush 向客户端返回 ResponseSample 对象，其中包含返回码、响应数据以及时间戳。

public class RequestSampleHandler extends ChannelInboundHandlerAdapter {

    @Override

    public void channelRead(ChannelHandlerContext ctx, Object msg) {

        String data = ((ByteBuf) msg).toString(CharsetUtil.UTF\_8);

        ResponseSample response = new ResponseSample("OK", data, System.currentTimeMillis());

        ctx.channel().writeAndFlush(response);

    }

}

通过以上的代码示例我们可以描绘出 Pipeline 的链表结构，如下图所示。



 RequestSampleHandler 调用 writeAndFlush方法传输数据

## writeAndFlush 的**核心逻辑**

private void write(Object msg, boolean flush, ChannelPromise promise) {

    // ...... 省略部分非核心代码 ......

    // 找到 Pipeline 链表中下一个 Outbound 类型的 ChannelHandler 节点

    final AbstractChannelHandlerContext next = findContextOutbound(flush ?

            (MASK\_WRITE | MASK\_FLUSH) : MASK\_WRITE);

    final Object m = pipeline.touch(msg, next);

    EventExecutor executor = next.executor();

    // 判断当前线程是否是 NioEventLoop 中的线程

    if (executor.inEventLoop()) {

        if (flush) {

            // 因为 flush == true，所以流程走到这里

            next.invokeWriteAndFlush(m, promise);

        } else {

            next.invokeWrite(m, promise);

        }

    } else {

        final AbstractWriteTask task;

        if (flush) {

            task = WriteAndFlushTask.newInstance(next, m, promise);

        }  else {

            task = WriteTask.newInstance(next, m, promise);

        }

        if (!safeExecute(executor, task, promise, m)) {

            task.cancel();

        }

    }

}

首先我们确认下方法的入参，因为我们需要执行 flush 动作，所以 flush == true；write 方法还需要 ChannelPromise 参数，可见写操作是个异步的过程。AbstractChannelHandlerContext 会默认初始化一个 ChannelPromise 完成该异步操作，ChannelPromise 内部持有当前的 Channel 和 EventLoop，此外你可以向 ChannelPromise 中注册回调监听 listener 来获得异步操作的结果。

write 方法的核心逻辑主要分为三个重要步骤，我已经以注释的形式在源码中标注出来。下面我们将结合上文中的 EchoServer 代码示例详细分析 write 方法的执行机制。

第一步，调用 findContextOutbound 方法找到 Pipeline 链表中下一个 Outbound 类型的 ChannelHandler。在我们模拟的场景中下一个 Outbound 节点是 ResponseSampleEncoder。

第二步，通过 inEventLoop 方法判断当前线程的身份标识，如果当前线程和 EventLoop 分配给当前 Channel 的线程是同一个线程的话，那么所提交的任务将被立即执行。否则当前的操作将被封装成一个 Task 放入到 EventLoop 的任务队列，稍后执行。所以 writeAndFlush 是否是线程安全的呢，你心里有答案了吗？

第三步，因为 flush== true，将会直接执行 next.invokeWriteAndFlush(m, promise) 这行代码，我们跟进去源码。发现最终会它会执行下一个 ChannelHandler 节点的 write 方法，那么流程又回到了 到 AbstractChannelHandlerContext 中重复执行 write 方法，继续寻找下一个 Outbound 节点。

private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {

    if (invokeHandler()) {

        invokeWrite0(msg, promise);

        invokeFlush0();

    } else {

        writeAndFlush(msg, promise);

    }

}

private void invokeWrite0(Object msg, ChannelPromise promise) {

    try {

        ((ChannelOutboundHandler) handler()).write(this, msg, promise);

    } catch (Throwable t) {

        notifyOutboundHandlerException(t, promise);

    }

}

调用 writeAndFlush 时数据是在 Outbound 类型的 ChannelHandler 节点之间进行传播，那么最终数据是如何写到 Socket 底层的呢？

## 写 Buffer 队列

据将会在 Pipeline 中一直寻找 Outbound 节点并向前传播，直到 Head 节点结束，由 Head 节点完成最后的数据发送。所以 Pipeline 中的 Head 节点在完成 writeAndFlush 过程中扮演着重要的角色。

// HeadContext # write

@Override

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) {

    unsafe.write(msg, promise);

}

// AbstractChannel # AbstractUnsafe # write

@Override

public final void write(Object msg, ChannelPromise promise) {

    assertEventLoop();

    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;

    if (outboundBuffer == null) {

        safeSetFailure(promise, newClosedChannelException(initialCloseCause));

        ReferenceCountUtil.release(msg);

        return;

    }

    int size;

    try {

        msg = filterOutboundMessage(msg); // 过滤消息

        size = pipeline.estimatorHandle().size(msg);

        if (size < 0) {

            size = 0;

        }

    } catch (Throwable t) {

        safeSetFailure(promise, t);

        ReferenceCountUtil.release(msg);

        return;

    }

    outboundBuffer.addMessage(msg, size, promise); // 向 Buffer 中添加数据

}

可以看出 Head 节点是通过调用 unsafe 对象完成数据写入的，unsafe 对应的是 NioSocketChannelUnsafe 对象实例，最终调用到 AbstractChannel 中的 write 方法，该方法有两个重要的点需要指出：

1. filterOutboundMessage 方法会对待写入的 msg 进行过滤，如果 msg 使用的不是 DirectByteBuf，那么它会将 msg 转换成 DirectByteBuf。
2. ChannelOutboundBuffer 可以理解为一个缓存结构，从源码最后一行 outboundBuffer.addMessage 可以看出是在向这个缓存中添加数据，所以 ChannelOutboundBuffer 才是理解数据发送的关键。

writeAndFlush 主要分为两个步骤，write 和 flush。通过上面的分析可以看出只调用 write 方法，数据并不会被真正发送出去，而是存储在 ChannelOutboundBuffer 的缓存内。下面我们重点分析一下 ChannelOutboundBuffer 的内部构造，跟进一下 addMessage 的源码：

public void addMessage(Object msg, int size, ChannelPromise promise) {

    Entry entry = Entry.newInstance(msg, size, total(msg), promise);

    if (tailEntry == null) {

        flushedEntry = null;

    } else {

        Entry tail = tailEntry;

        tail.next = entry;

    }

    tailEntry = entry;

    if (unflushedEntry == null) {

        unflushedEntry = entry;

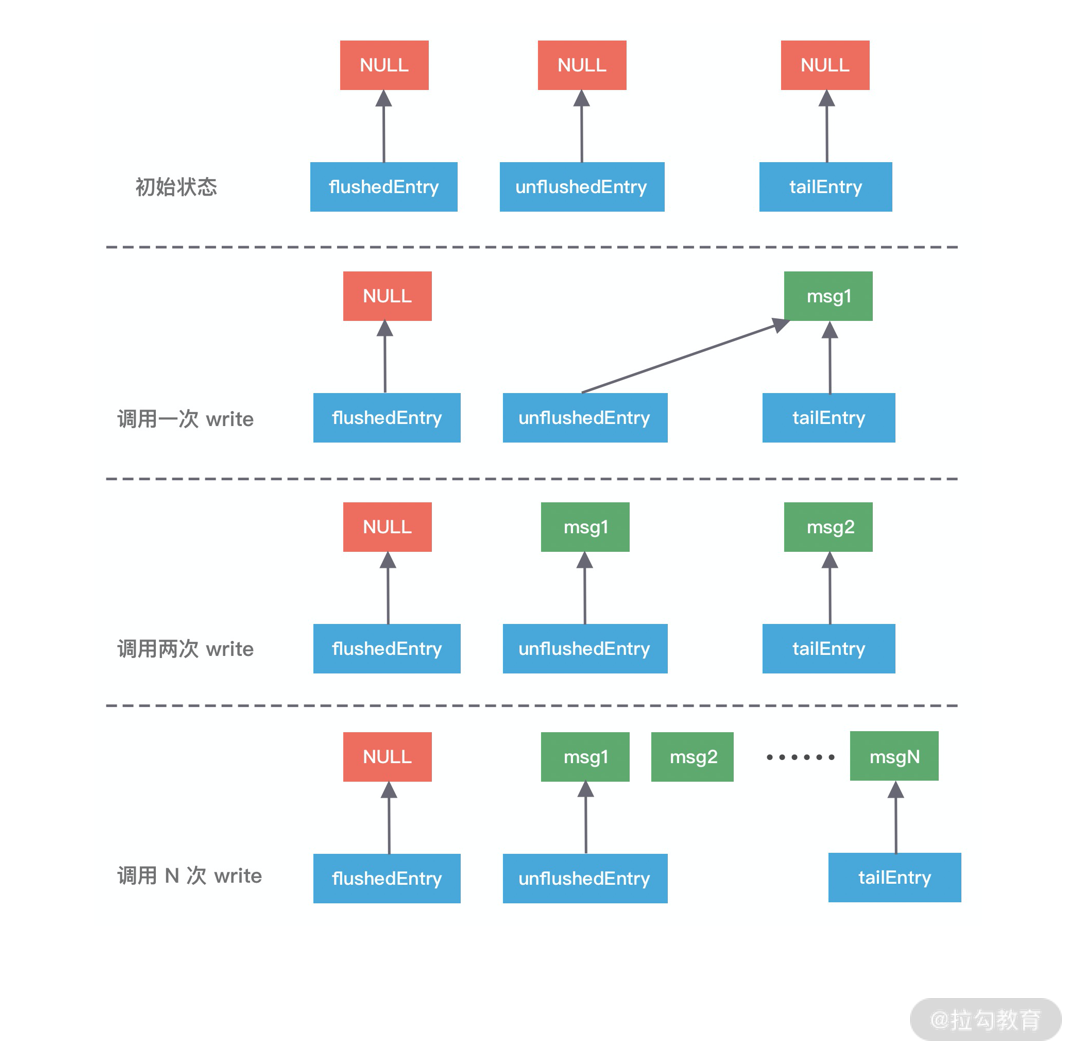
    }

    incrementPendingOutboundBytes(entry.pendingSize, false);

}

ChannelOutboundBuffer 缓存是一个链表结构，每次传入的数据都会被封装成一个 Entry 对象添加到链表中。ChannelOutboundBuffer 包含**三个非常重要的指针**：第一个被写到缓冲区的**节点 flushedEntry**、第一个未被写到缓冲区的**节点 unflushedEntry**和最后一个**节点 tailEntry。**

在初始状态下这三个指针都指向 NULL，当我们每次调用 write 方法是，都会调用 addMessage 方法改变这三个指针的指向，可以参考下图理解指针的移动过程会更加形象。



第一次调用 write，因为链表里只有一个数据，所以 unflushedEntry 和 tailEntry 指针都指向第一个添加的数据 msg1。flushedEntry 指针在没有触发 flush 动作时会一直指向 NULL。

第二次调用 write，tailEntry 指针会指向新加入的 msg2，unflushedEntry 保持不变。

第 N 次调用 write，tailEntry 指针会不断指向新加入的 msgN，unflushedEntry 依然保持不变，unflushedEntry 和 tailEntry 指针之间的数据都是未写入 Socket 缓冲区的。

以上便是写 Buffer 队列写入数据的实现原理，但是我们不可能一直向缓存中写入数据，所以 addMessage 方法中每次写入数据后都会调用 incrementPendingOutboundBytes 方法判断缓存的水位线，具体源码如下。

private static final int DEFAULT\_LOW\_WATER\_MARK = 32 \* 1024;

private static final int DEFAULT\_HIGH\_WATER\_MARK = 64 \* 1024;

private void incrementPendingOutboundBytes(long size, boolean invokeLater) {

    if (size == 0) {

        return;

    }

    long newWriteBufferSize = TOTAL\_PENDING\_SIZE\_UPDATER.addAndGet(this, size);

    // 判断缓存大小是否超过高水位线

    if (newWriteBufferSize > channel.config().getWriteBufferHighWaterMark()) {

        setUnwritable(invokeLater);

    }

}

incrementPendingOutboundBytes 的逻辑非常简单，每次添加数据时都会累加数据的字节数，然后判断缓存大小是否超过所设置的高水位线 64KB，如果超过了高水位，那么 Channel 会被设置为不可写状态。直到缓存的数据大小低于低水位线 32KB 以后，Channel 才恢复成可写状态。

有关写数据的逻辑已经分析完了，那么执行 flush 动作缓存又会是什么变化呢？我们接下来一起看下 flush 的工作原理吧。

## 刷新 Buffer 队列

当执行完 write 写操作之后，invokeFlush0 会触发 flush 动作，与 write 方法类似，flush 方法同样会从 Tail 节点开始传播到 Head 节点，同样我们跟进下 HeadContext 的 flush 源码：

// HeadContext # flush

@Override

public void flush(ChannelHandlerContext ctx) {

    unsafe.flush();

}

// AbstractChannel # flush

@Override

public final void flush() {

    assertEventLoop();

    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;

    if (outboundBuffer == null) {

        return;

    }

    outboundBuffer.addFlush();

    flush0();

}

可以看出 flush 的核心逻辑主要分为两个步骤：addFlush 和 flush0，下面我们逐一对它们进行分析。

首先看下 addFlush 方法的源码：

// ChannelOutboundBuffer # addFlush

public void addFlush() {

    Entry entry = unflushedEntry;

    if (entry != null) {

        if (flushedEntry == null) {

            flushedEntry = entry;

        }

        do {

            flushed ++;

            if (!entry.promise.setUncancellable()) {

                int pending = entry.cancel();

                // 减去待发送的数据，如果总字节数低于低水位，那么 Channel 将变为可写状态

                decrementPendingOutboundBytes(pending, false, true);

            }

            entry = entry.next;

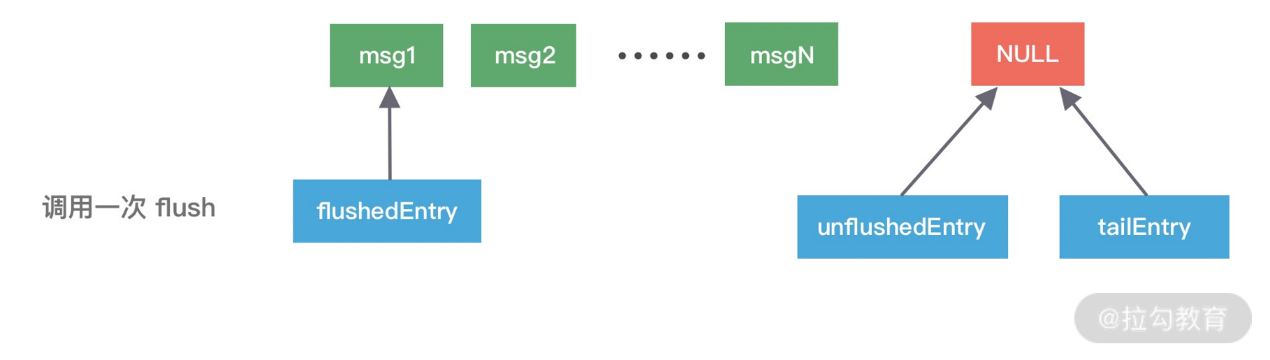
        } while (entry != null);

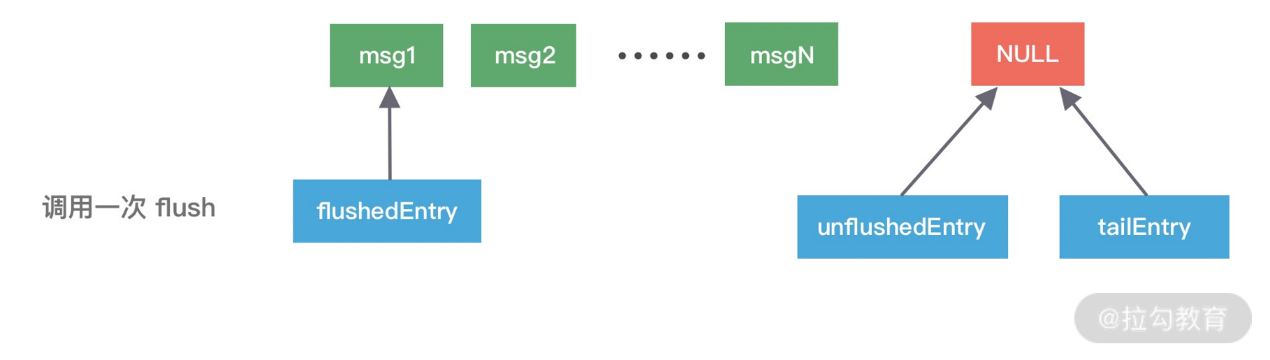
        unflushedEntry = null;

    }

}

addFlush 方法同样也会操作 ChannelOutboundBuffer 缓存数据。在执行 addFlush 方法时，缓存中的指针变化又是如何呢？如下图所示，我们在写入流程的基础上继续进行分析。





此时 flushedEntry 指针有所改变，变更为 unflushedEntry 指针所指向的数据，然后 unflushedEntry 指针指向 NULL，flushedEntry 指针指向的数据才会被真正发送到 Socket 缓冲区。

在 addFlush 源码中 decrementPendingOutboundBytes 与之前 addMessage 源码中的 incrementPendingOutboundBytes 是相对应的。decrementPendingOutboundBytes 主要作用是减去待发送的数据字节，如果缓存的大小已经小于低水位，那么 Channel 会恢复为可写状态。

addFlush 的大体流程我们已经介绍完毕，接下来便是第二步负责发送数据的 flush0 方法。同样我们跟进 flush0 的源码，定位出 flush0 的核心调用链路：

// AbstractNioUnsafe # flush0

@Override

protected final void flush0() {

    if (!isFlushPending()) {

        super.flush0();

    }

}

// AbstractNioByteChannel # doWrite

@Override

protected void doWrite(ChannelOutboundBuffer in) throws Exception {

    int writeSpinCount = config().getWriteSpinCount();

    do {

        Object msg = in.current();

        if (msg == null) {

            clearOpWrite();

            return;

        }

        writeSpinCount -= doWriteInternal(in, msg);

    } while (writeSpinCount > 0);

    incompleteWrite(writeSpinCount < 0);

}

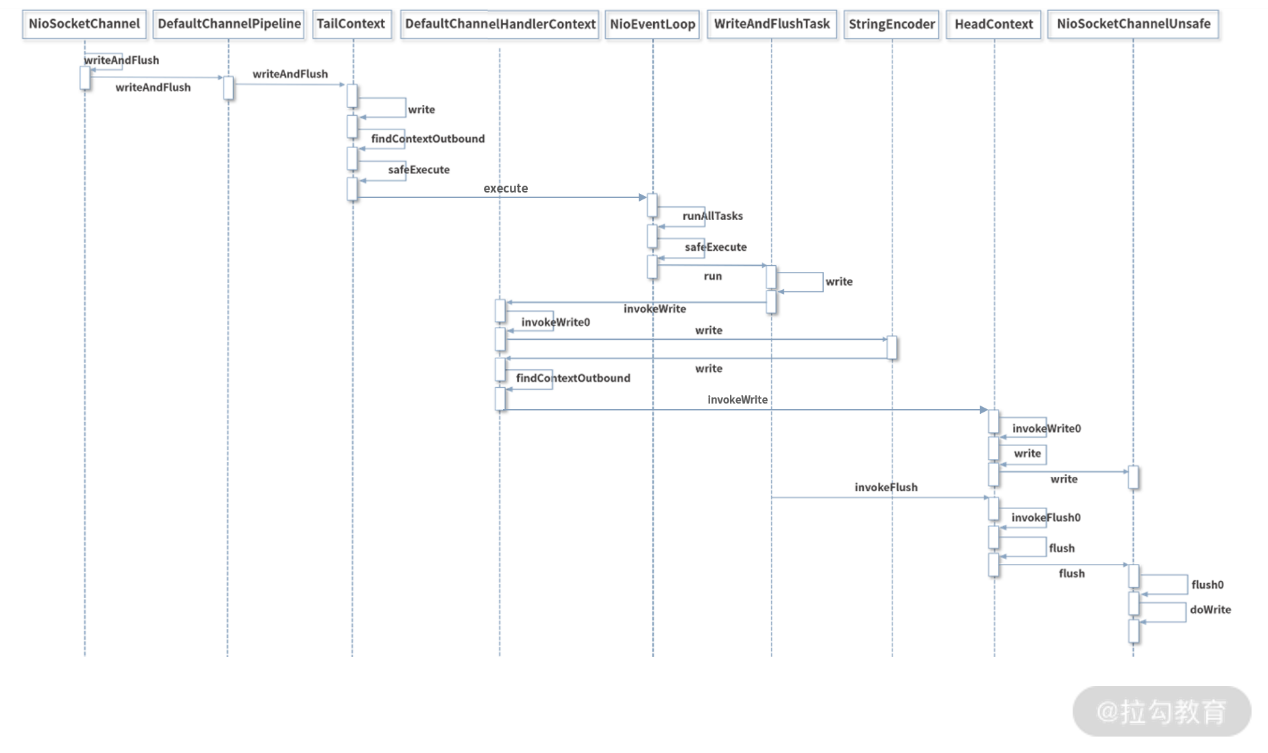
实际 flush0 的调用层次很深，但其实核心的逻辑在于 AbstractNioByteChannel 的 doWrite 方法，该方法负责将数据真正写入到 Socket 缓冲区。doWrite 方法的处理流程主要分为三步：

第一，根据配置获取自旋锁的次数 writeSpinCount。那么你的疑问就来了，这个自旋锁的次数主要是用来干什么的呢？当我们向 Socket 底层写数据的时候，如果每次要写入的数据量很大，是不可能一次将数据写完的，所以只能分批写入。Netty 在不断调用执行写入逻辑的时候，EventLoop 线程可能一直在等待，这样有可能会阻塞其他事件处理。所以这里自旋锁的次数相当于控制一次写入数据的最大的循环执行次数，如果超过所设置的自旋锁次数，那么写操作将会被暂时中断。

第二，根据自旋锁次数重复调用 doWriteInternal 方法发送数据，每成功发送一次数据，自旋锁的次数 writeSpinCount 减 1，当 writeSpinCount 耗尽，那么 doWrite 操作将会被暂时中断。doWriteInternal 的源码涉及 JDK NIO 底层，在这里我们不再深入展开，它的主要作用在于删除缓存中的链表节点以及调用底层 API 发送数据，有兴趣的同学可以自行研究。

第三，调用 incompleteWrite 方法确保数据能够全部发送出去，因为自旋锁次数的限制，可能数据并没有写完，所以需要继续 OP\_WRITE 事件；如果数据已经写完，清除 OP\_WRITE 事件即可。

至此，整个 writeAndFlush 的工作原理已经全部分析完了，整个过程的调用层次比较深，我整理了 writeAndFlush 的时序图，如下所示，帮助大家梳理 writeAndFlush 的调用流程，加深对上述知识点的理解。



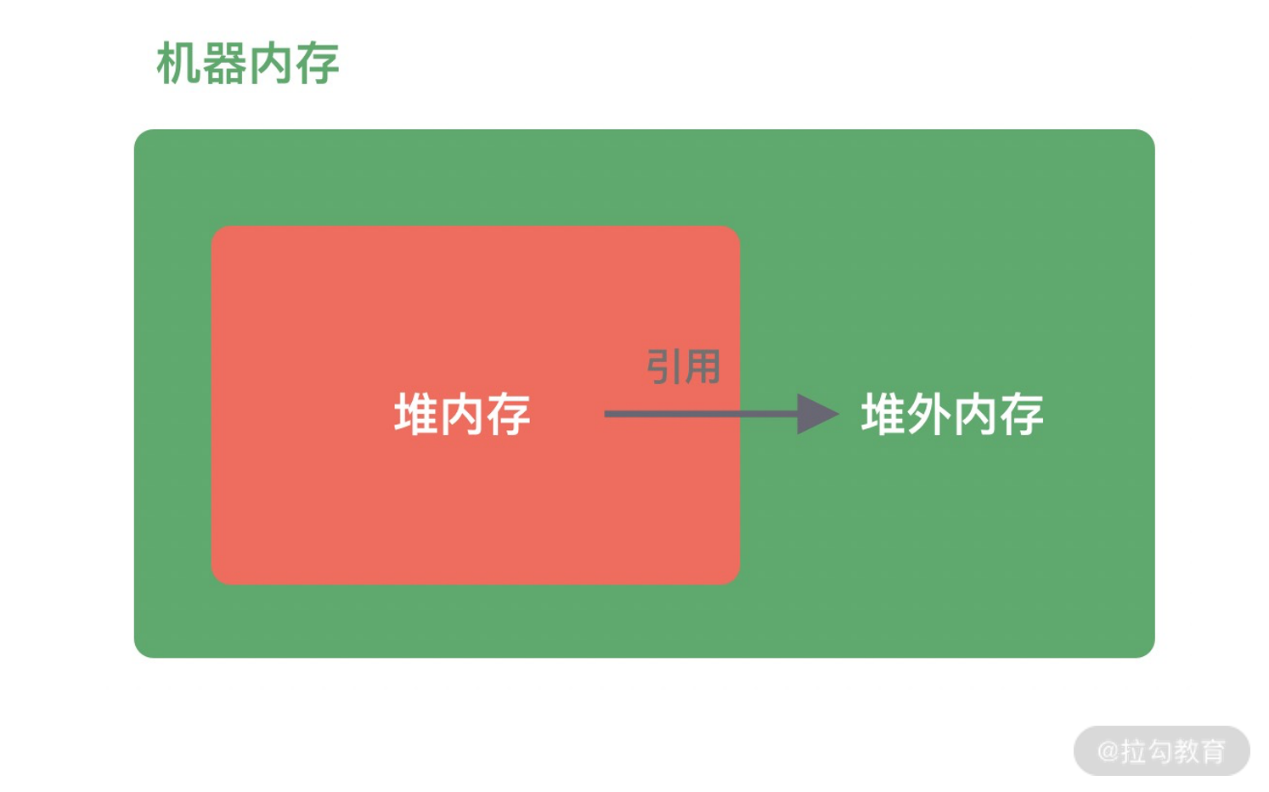
* writeAndFlush 属于出站操作，它是从 Pipeline 的 Tail 节点开始进行事件传播，一直向前传播到 Head 节点。不管在 write 还是 flush 过程，Head 节点都中扮演着重要的角色。
* write 方法并没有将数据写入 Socket 缓冲区，只是将数据写入到 ChannelOutboundBuffer 缓存中，ChannelOutboundBuffer 缓存内部是由单向链表实现的。
* flush 方法才最终将数据写入到 Socket 缓冲区。

# 堆外内存

## 为什么需要堆外内存

在 Java 中对象都是在堆内分配的，通常我们说的**JVM 内存**也就指的**堆内内存**，**堆内内存**完全被**JVM 虚拟机**所管理，JVM 有自己的垃圾回收算法，对于使用者来说不必关心对象的内存如何回收。

**堆外内存**与堆内内存相对应，对于整个机器内存而言，除**堆内内存以外部分即为堆外内存**，如下图所示。堆外内存不受 JVM 虚拟机管理，直接由操作系统管理。



堆外内存和堆内内存各有利弊，这里我针对其中重要的几点进行说明。

1. 堆内内存由 JVM GC 自动回收内存，降低了 Java 用户的使用心智，但是 GC 是需要时间开销成本的，堆外内存由于不受 JVM 管理，所以在一定程度上可以降低 GC 对应用运行时带来的影响。
2. 堆外内存需要手动释放，这一点跟 C/C++ 很像，稍有不慎就会造成应用程序内存泄漏，当出现内存泄漏问题时排查起来会相对困难。
3. 当进行网络 I/O 操作、文件读写时，堆内内存都需要转换为堆外内存，然后再与底层设备进行交互，这一点在介绍 writeAndFlush 的工作原理中也有提到，所以直接使用堆外内存可以减少一次内存拷贝。
4. 堆外内存可以实现进程之间、JVM 多实例之间的数据共享。

由此可以看出，如果你想实现高效的 I/O 操作、缓存常用的对象、降低 JVM GC 压力，堆外内存是一个非常不错的选择。

## 堆外内存的分配

Java 中堆外内存的分配方式有两种：**ByteBuffer#allocateDirect**和**Unsafe#allocateMemory**。

 Java NIO 包中的 ByteBuffer 类的分配方式，使用方式如下：

// 分配 10M 堆外内存

ByteBuffer buffer = ByteBuffer.allocateDirect(10 \* 1024 \* 1024);

跟进 ByteBuffer.allocateDirect 源码，发现其中直接调用的 DirectByteBuffer 构造函数

DirectByteBuffer(int cap) {

    super(-1, 0, cap, cap);

    boolean pa = VM.isDirectMemoryPageAligned();

    int ps = Bits.pageSize();

    long size = Math.max(1L, (long)cap + (pa ? ps : 0));

    Bits.reserveMemory(size, cap);

    long base = 0;

    try {

        base = unsafe.allocateMemory(size);

    } catch (OutOfMemoryError x) {

        Bits.unreserveMemory(size, cap);

        throw x;

    }

    unsafe.setMemory(base, size, (byte) 0);

    if (pa && (base % ps != 0)) {

        address = base + ps - (base & (ps - 1));

    } else {

        address = base;

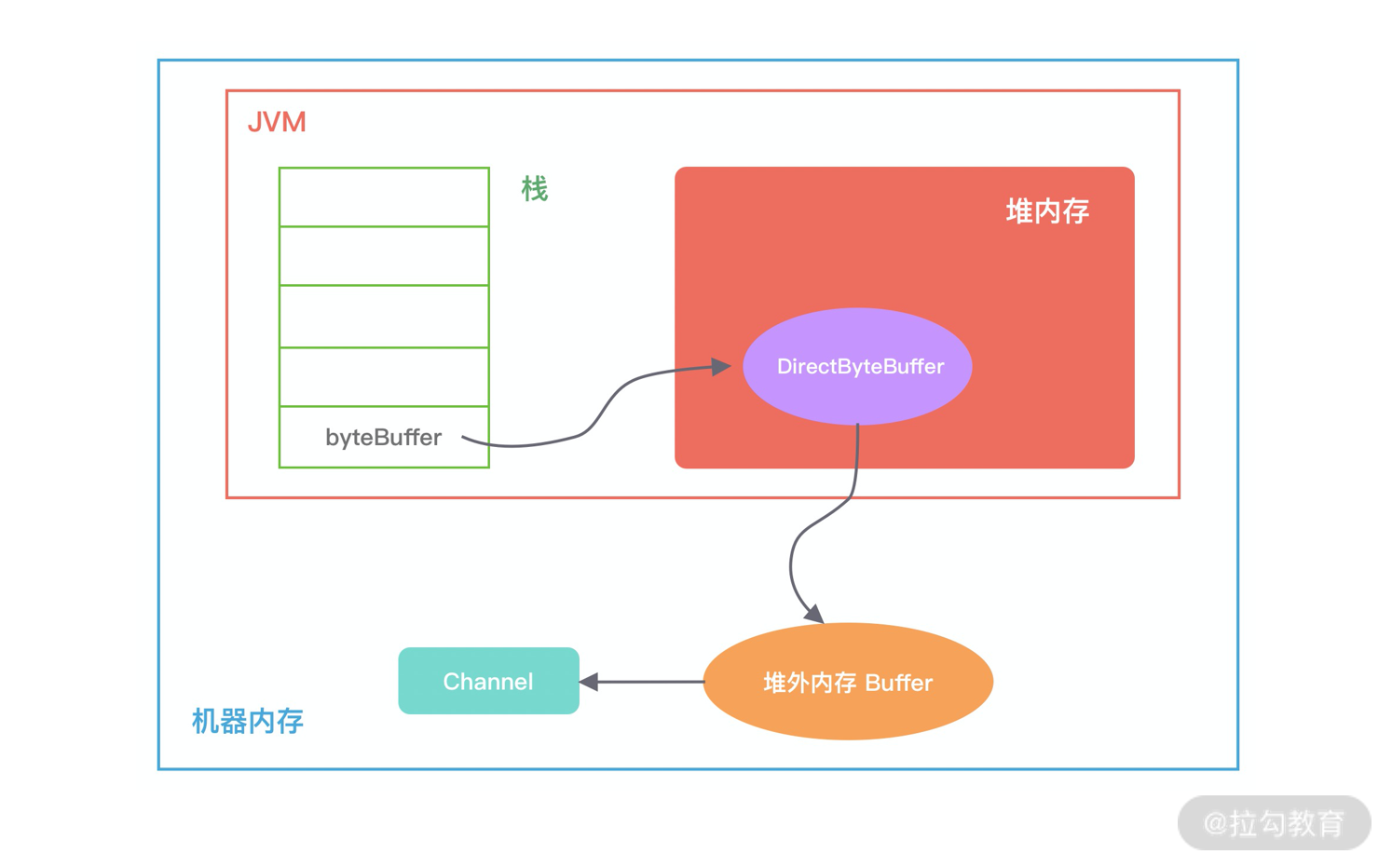
    }

    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));

    att = null;

}

描述了 DirectByteBuffer 的内存引用情况，方便你更好地理解上述源码的初始化过程。在堆内存放的 DirectByteBuffer 对象并不大，仅仅包含堆外内存的地址、大小等属性，同时还会创建对应的 Cleaner 对象，通过 ByteBuffer 分配的堆外内存不需要手动回收，它可以被 JVM 自动回收。当堆内的 DirectByteBuffer 对象被 GC 回收时，Cleaner 就会用于回收对应的堆外内存。



从 DirectByteBuffer 的构造函数中可以看出，真正分配堆外内存的逻辑还是通过 **unsafe.allocateMemory(size)，**

Unsafe 是一个非常不安全的类，它用于执行内存访问、分配、修改等敏感操作，可以越过 JVM 限制的枷锁。Unsafe 最初并不是为开发者设计的，使用它时虽然可以获取对底层资源的控制权，但也失去了安全性的保证，所以使用 Unsafe 一定要慎重。Netty 中依赖了 Unsafe 工具类，是因为 Netty 需要与底层 Socket 进行交互，Unsafe 在提升 Netty 的性能方面起到了一定的帮助。

在 Java 中是不能直接使用 Unsafe 的，但是我们可以通过反射获取 Unsafe 实例，使用方式如下所示。

private static Unsafe unsafe = null;

static {

    try {

        Field getUnsafe = Unsafe.class.getDeclaredField("theUnsafe");

        getUnsafe.setAccessible(true);

        unsafe = (Unsafe) getUnsafe.get(null);

    } catch (NoSuchFieldException | IllegalAccessException e) {

        e.printStackTrace();

    }

}

获得 Unsafe 实例后，我们可以通过 allocateMemory 方法分配堆外内存，allocateMemory 方法返回的是内存地址，使用方法如下所示：

// 分配 10M 堆外内存

long address = unsafe.allocateMemory(10 \* 1024 \* 1024);

与 DirectByteBuffer 不同的是，Unsafe#allocateMemory 所分配的内存必须自己手动释放，否则会造成内存泄漏，这也是 Unsafe 不安全的体现。Unsafe 同样提供了内存释放的操作：

unsafe.freeMemory(address);

对于 Java 开发者而言，常用的是 ByteBuffer.allocateDirect 分配方式，我们平时常说的堆外内存泄漏都与该分配方式有关，接下来我们一起看看使用 ByteBuffer 分配的堆外内存如何被 JVM 回收，这对我们排查堆外内存泄漏问题有较大的帮助。

## 堆外内存的回收

因为 DirectByteBuffer 对象有可能长时间存在于堆内内存，所以它很可能晋升到 JVM 的老年代，所以这时候 DirectByteBuffer 对象的回收需要依赖 Old GC 或者 Full GC 才能触发清理。如果长时间没有 Old GC 或者 Full GC 执行，那么堆外内存即使不再使用，也会一直在占用内存不释放，很容易将机器的物理内存耗尽，这是相当危险的。

因为 JVM 并不知道堆外内存是不是已经不足了，所以我们最好通过 JVM 参数 -XX:MaxDirectMemorySize 指定堆外内存的上限大小，当堆外内存的大小超过该阈值时，就会触发一次 Full GC 进行清理回收，如果在 Full GC 之后还是无法满足堆外内存的分配，那么程序将会抛出 OOM 异常。

此外在 ByteBuffer.allocateDirect 分配的过程中，如果没有足够的空间分配堆外内存，在 Bits.reserveMemory 方法中也会主动调用 System.gc() 强制执行 Full GC，但是在生产环境一般都是设置了 -XX:+DisableExplicitGC，System.gc() 是不起作用的，所以依赖 System.gc() 并不是一个好办法。

DirectByteBuffer 在初始化时会创建一个 Cleaner 对象，它会负责堆外内存的回收工作

Java 对象有四种引用方式：强引用 StrongReference、软引用 SoftReference、弱引用 WeakReference 和虚引用 PhantomReference。其中 PhantomReference 是最不常用的一种引用方式，**Cleaner 就属于 PhantomReference 的子类**，如以下源码所示，PhantomReference 不能被单独使用，需要与引用队列 ReferenceQueue 联合使用。

public class Cleaner extends java.lang.ref.PhantomReference<java.lang.Object> {

    private static final java.lang.ref.ReferenceQueue<java.lang.Object> dummyQueue;

    private static sun.misc.Cleaner first;

    private sun.misc.Cleaner next;

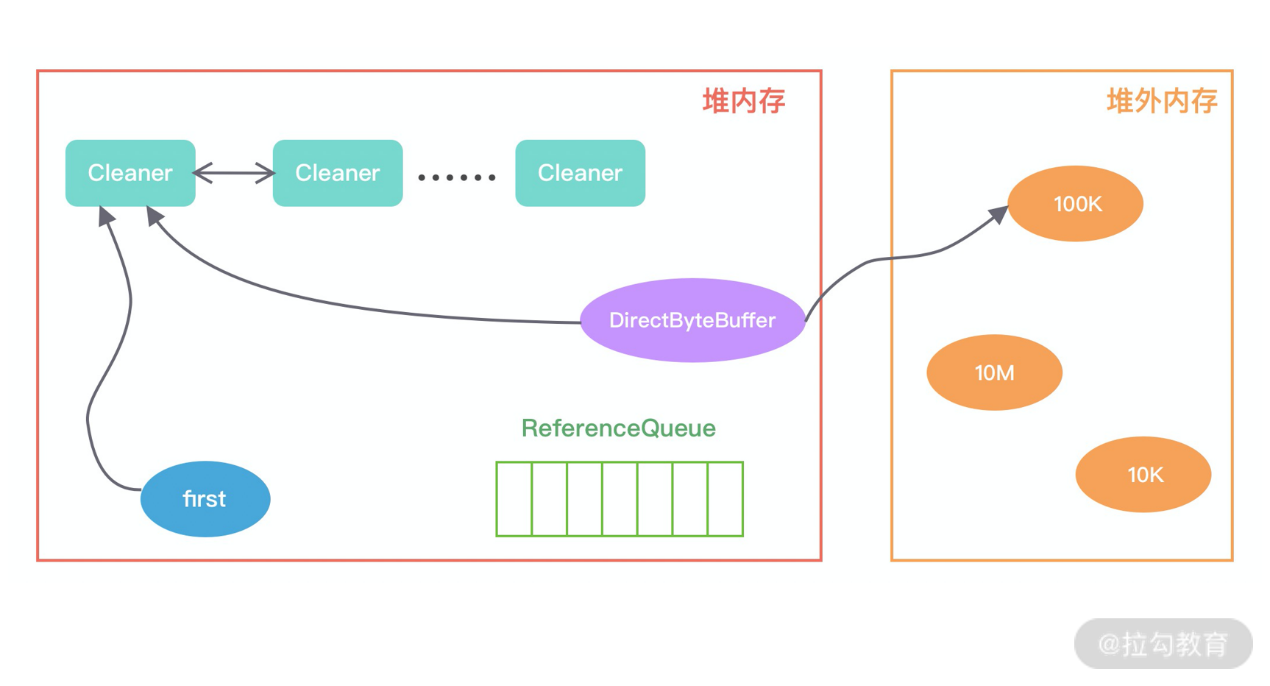
    private sun.misc.Cleaner prev;

    private final java.lang.Runnable thunk;

    public void clean() {}

}

当初始化堆外内存时，内存中的对象引用情况如下图所示，first 是 Cleaner 类中的静态变量，Cleaner 对象在初始化时会加入 Cleaner 链表中。DirectByteBuffer 对象包含堆外内存的地址、大小以及 Cleaner 对象的引用，ReferenceQueue 用于保存需要回收的 Cleaner 对象。



当发生 GC 时，DirectByteBuffer 对象被回收，内存中的对象引用情况发生了如下变化：

此时 Cleaner 对象不再有任何引用关系，在下一次 GC 时，该 Cleaner 对象将被添加到 ReferenceQueue 中，并执行 clean() 方法。clean() 方法主要做两件事情：

1. 将 Cleaner 对象从 Cleaner 链表中移除；
2. 调用 unsafe.freeMemory 方法清理堆外内存。

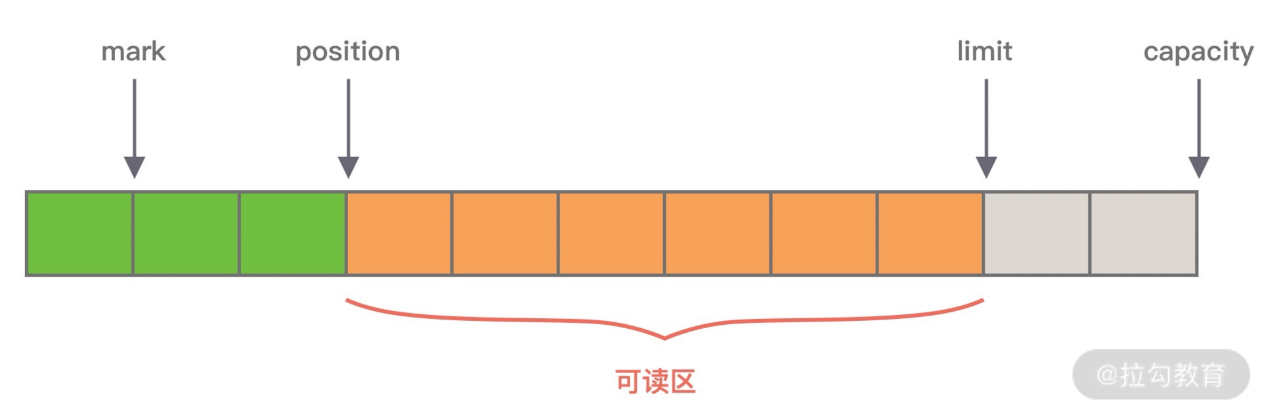
堆外内存是一把双刃剑，在网络 I/O、文件读写、分布式缓存等领域使用堆外内存都更加简单、高效，此外使用堆外内存不受 JVM 约束，可以避免 JVM GC 的压力，降低对业务应用的影响。当然天下没有免费的午餐，堆外内存也不能滥用，使用堆外内存你就需要关注内存回收问题，虽然 JVM 在一定程度上帮助我们实现了堆外内存的自动回收，但我们仍然需要培养类似 C/C++ 的分配/回收的意识，出现内存泄漏问题能够知道如何分析和处理。

# Netty 数据传输载体 ByteBuf 详解

Netty 大量使用了自己实现的 ByteBuf 工具类，ByteBuf 是 Netty 的数据容器，所有网络通信中字节流的传输都是通过 ByteBuf 完成的。

## NIO 的 ByteBuffe

ByteBuffer 的内部结构



ByteBuffer 包含以下四个基本属性：

mark：为某个读取过的关键位置做标记，方便回退到该位置；

position：当前读取的位置；

limit：buffer 中有效的数据长度大小；

capacity：初始化时的空间容量。

以上四个基本属性的关系是：mark <= position <= limit <= capacity。

Bytebuffer的缺陷

1. ByteBuffer 分配的长度是固定的，无法动态扩缩容，所以很难控制需要分配多大的容量。**如果分配太大容量**，容易造成内存浪费；**如果分配太小**，存放太大的数据会抛出 BufferOverflowException 异常。在使用 ByteBuffer 时，为了避免容量不足问题，你**必须每次在存放数据的时候对容量大小做校验**，**如果超出 ByteBuffer 最大容量，那么需要重新开辟一个更大容量的 ByteBuffer，将已有的数据迁移过去**。**整个过程相对烦琐，对开发者而言是非常不友好的**。
2. ByteBuffer 只能通过 position 获取当前可操作的位置，因为读写共用的 position 指针，**所以需要频繁调用 flip、rewind 方法切换读写状态**，开发者必须很小心处理 ByteBuffer 的数据读写，稍不留意就会出错。

ByteBuffer 作为网络通信中高频使用的数据载体，显然不能够满足 Netty 的需求，Netty 重新实现了一个性能更高、易用性更强的 ByteBuf，相比于 ByteBuffer 它提供了很多非常酷的特性：

容量可以按需动态扩展，类似于 StringBuffer；

读写采用了不同的指针，读写模式可以随意切换，不需要调用 flip 方法；

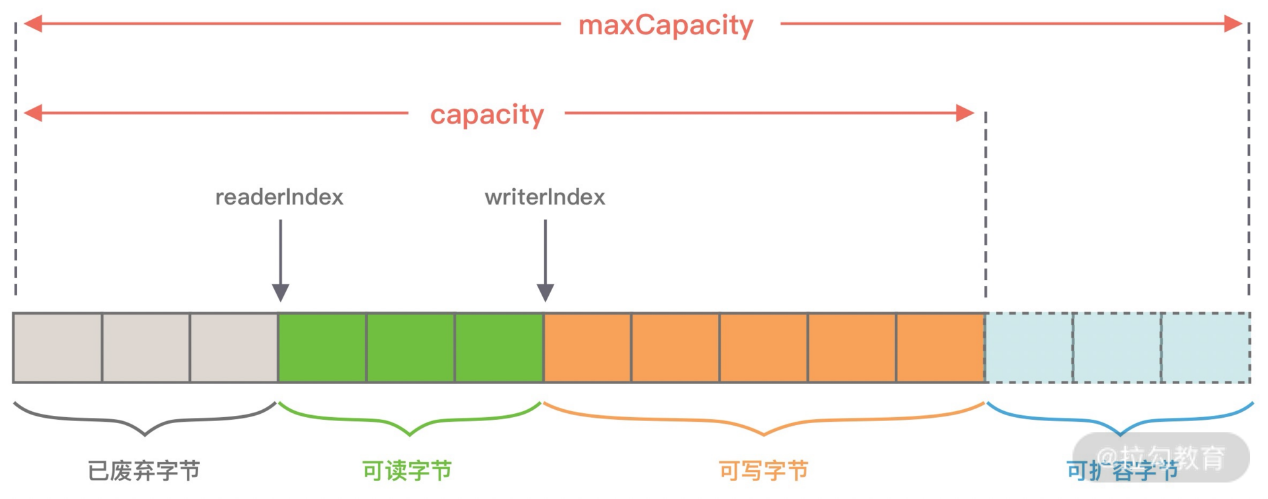
通过内置的复合缓冲类型可以实现零拷贝；

支持引用计数；

支持缓存池。

## ByteBuf

### ByteBuf 内部结构



ByteBuf 包含三个指针：读指针 readerIndex、写指针 writeIndex、最大容量 maxCapacity，根据指针的位置又可以将 ByteBuf 内部结构可以分为四个部分：

第一部分是废弃字节，表示已经丢弃的无效字节数据。

第二部分是可读字节，表示 ByteBuf 中可以被读取的字节内容，可以通过 writeIndex - readerIndex 计算得出。从 ByteBuf 读取 N 个字节，readerIndex 就会自增 N，readerIndex 不会大于 writeIndex，当 readerIndex == writeIndex 时，表示 ByteBuf 已经不可读。

第三部分是可写字节，向 ByteBuf 中写入数据都会存储到可写字节区域。向 ByteBuf 写入 N 字节数据，writeIndex 就会自增 N，当 writeIndex 超过 capacity，表示 ByteBuf 容量不足，需要扩容。

第四部分是可扩容字节，表示 ByteBuf 最多还可以扩容多少字节，当 writeIndex 超过 capacity 时，会触发 ByteBuf 扩容，最多扩容到 maxCapacity 为止，超过 maxCapacity 再写入就会出错。

### 引用计数

ByteBuf 是基于引用计数设计的，它实现了 ReferenceCounted 接口，ByteBuf 的生命周期是由引用计数所管理。只要引用计数大于 0，表示 ByteBuf 还在被使用；当 ByteBuf 不再被其他对象所引用时，引用计数为 0，那么代表该对象可以被释放。

引用计数对于 Netty 设计缓存池化有非常大的帮助，当引用计数为 0，该 ByteBuf 可以被放入到对象池中，避免每次使用 ByteBuf 都重复创建，对于实现高性能的内存管理有着很大的意义。

### ByteBuf 分类

Heap/Direct 就是堆内和堆外内存。Heap 指的是在 JVM 堆内分配，底层依赖的是字节数据；Direct 则是堆外内存，不受 JVM 限制，分配方式依赖 JDK 底层的 ByteBuffer。

Pooled/Unpooled 表示池化还是非池化内存。Pooled 是从预先分配好的内存中取出，使用完可以放回 ByteBuf 内存池，等待下一次分配。而 Unpooled 是直接调用系统 API 去申请内存，确保能够被 JVM GC 管理回收。

Unsafe/非 Unsafe 的区别在于操作方式是否安全。 Unsafe 表示每次调用 JDK 的 Unsafe 对象操作物理内存，依赖 offset + index 的方式操作数据。非 Unsafe 则不需要依赖 JDK 的 Unsafe 对象，直接通过数组下标的方式操作数据。

# 高性能内存分配器 jemalloc 基本原理

jemalloc 是由 Jason Evans 在 FreeBSD 项目中引入的新一代内存分配器。它是一个通用的 malloc 实现，侧重于减少内存碎片和提升高并发场景下内存的分配效率，其目标是能够替代 malloc。jemalloc 应用十分广泛，在 Firefox、Redis、Rust、Netty 等出名的产品或者编程语言中都有大量使用。

## 常用内存分配算法

# Netty 高性能内存管理设计